FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Review of Machine Learning Deployment Frameworks

Alexandre Araújo Pires Mourão

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Canas Ferreira

July 23, 2024

Resumo

No domínio em rápida evolução dos serviços de mobilidade, a implementação eficiente de modelos de aprendizagem automática em ambientes com recursos limitados, como os encontrados em aplicações IoT, continua a ser um desafio significativo. Esta dissertação centra-se na avaliação e comparação de várias estruturas de aprendizagem automática, especificamente TensorFlow Lite, ONNXRuntime e PyTorch, para identificar a estrutura mais adequada para uma implementação eficiente em plataformas de hardware heterogéneas normalmente utilizadas em sistemas IoT. O objetivo principal é facilitar a transição dos modelos de aprendizagem automática do desenvolvimento para a produção, garantindo a compatibilidade, a eficiência computacional, a estabilidade e a segurança em várias configurações de hardware.

A investigação envolve uma avaliação comparativa detalhada das estruturas seleccionadas, centrando-se nos principais indicadores de desempenho, como os tempos de inferência, a utilização de memória e a facilidade de implementação. Além disso, o estudo examina o impacto da complexidade do modelo na eficiência da implementação e explora técnicas de otimização, como a quantização, para melhorar o desempenho em dispositivos periféricos. Uma parte significativa do estudo inclui a análise do modelo de deteção de colisões da Bosch, avaliando o seu desempenho em diferentes estruturas para fornecer uma compreensão abrangente das capacidades de cada estrutura.

Através desta avaliação sistemática, a dissertação tem como objetivo oferecer informações valiosas sobre as melhores práticas para a implementação de modelos de aprendizagem automática em ambientes com recursos limitados. Os resultados enfatizam a importância de selecionar estruturas e técnicas de otimização adequadas para obter soluções de IAoT eficientes, escaláveis e robustas, melhorando, em última análise, a segurança e o conforto dos passageiros no sector da mobilidade.

Abstract

In the rapidly evolving field of mobility services, deploying machine learning models efficiently in resource-constrained environments, such as those found in IoT applications, remains a significant challenge. This dissertation focuses on evaluating and comparing several machine learning frameworks, specifically TensorFlow Lite, ONNXRuntime, and PyTorch, to identify the most suitable framework for efficient deployment on heterogeneous hardware platforms commonly used in IoT systems. The primary objective is to facilitate the transition of machine learning models from development to production, ensuring compatibility, computational efficiency, stability, and security across various hardware configurations.

The research involves detailed benchmarking of the selected frameworks, focusing on key performance indicators such as inference times, memory usage, and ease of deployment. Additionally, the study examines the impact of model complexity on deployment efficiency and explores optimization techniques such as quantization to enhance performance on edge devices. A significant part of the study includes the analysis of the Bosch collision detection model, evaluating its performance across different frameworks to provide a comprehensive understanding of each framework's capabilities.

Through this systematic evaluation, the dissertation aims to offer valuable insights into best practices for deploying machine learning models in resource-constrained environments. The findings emphasize the importance of selecting appropriate frameworks and optimization techniques to achieve efficient, scalable, and robust AIoT solutions, ultimately enhancing passenger safety and comfort in the mobility sector.

Sustainable Development Goals (SDGs)

This dissertation evaluates and compares machine learning frameworks like TensorFlow Lite, ONNXRuntime, and PyTorch to identify the best framework for efficient deployment on heterogeneous IoT hardware. The research is crucial for technological innovation and industrial enhancement, particularly in resource-constrained environments. It also focuses on improving passenger safety and comfort in mobility services, contributing to sustainable urban development.

By aligning with SDG 9 (Industry, Innovation, and Infrastructure), SDG 11 (Sustainable Cities and Communities), SDG 12 (Responsible Consumption and Production), and SDG 13 (Climate Action), the dissertation underscores the transformative potential of advanced machine learning frameworks. It emphasizes selecting appropriate frameworks and optimization techniques to achieve efficient, scalable, and robust AIoT solutions, contributing to the development of safer, more efficient, and sustainable transportation systems.

By focusing on these SDGs, the dissertation underscores the role of engineering and technological innovation in addressing critical global challenges, promoting sustainable development, and improving the quality of life in urban environments.

Table 1: SDG Contributions of the Dissertation

SDG	**Target**	**How the target is met**
9	9.5	The dissertation provides a systematic evaluation of best practices for deploying machine learning models in resource-constrained environments, offering insights to improve the efficiency and scalability of AIoT solutions, contributing to advanced industrial technologies.
	9.4	The dissertation promotes the use of efficient machine learning frameworks that optimize computational resources, reducing energy consumption and promoting environmentally sound technologies.
11	11.6	By optimizing IoT devices used in urban transportation systems, the dissertation helps in reducing energy consumption and waste, thereby minimizing the environmental impact of cities.
12	12.5	The dissertation's focus on optimizing ML models for efficiency reduces the need for frequent hardware upgrades, thereby reducing electronic waste and promoting sustainable consumption.
13	13.1	By promoting efficient and scalable AIoT solutions, the dissertation indirectly supports the development of technologies that can enhance resilience and adaptive capacity to climate-related impacts, particularly in urban environments.

Acknowledgements

I would like to express my deepest gratitude to Prof. João Canas Ferreira for his guidance, teaching, and patience throughout this Master's Thesis process. His expertise and support were invaluable in navigating the challenges of this academic journey.

My sincere thanks go to João Fernandes from Bosch for supervising my work and for his insightful suggestions and encouragement. I am also grateful to Bruno Faria and the entire team at Bosch for their warm welcome and support during my six-month internship. Their expertise and camaraderie made this experience incredibly enriching.

I am profoundly thankful to my family, especially my mother and father, for their unwavering support and encouragement throughout this journey. Their belief in me and constant support provided the foundation for my academic endeavors. I also extend my gratitude to my cousins, who helped me understand what university life entails and provided valuable guidance.

To my beloved girlfriend Kika, your support and love helped me through the toughest times. Your patience and encouragement were crucial to my success, and I am eternally grateful for your presence in my life.

I would also like to thank my university friends, the "Gang," for all the wonderful moments we shared at the Faculty of Engineering, University of Porto. Your friendship made this journey more enjoyable and memorable. A special thanks to my friend Pan, who has been a constant companion since the first day of university and shared this six-month journey with me at Bosch. Your friendship and support were invaluable.

Concluding, I want to thank all my friends for their support and for providing me with moments of happiness and joy. To all of you, I express my sincere gratitude.

Alexandre Araújo Pires Mourão

"And if you listen very hard, the tune will come to you at last."

Led Zeppelin, "Stairway to Heaven"

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ADD	Adition
AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
AOT	Ahead of Time
ARM	Advanced RISC Machine
CPU	Central Processing Unit
DFT	Discrete Forrier Transform
ECU	Electronic Control Units
FFT	Fast Forrier Transform
FLOP	Floating Point Operations Per Second
GPU	Graphic Processing Unit
IMU	Inertial Measurement Unit
IoT	Internet of Things
IoV	Internet of Vehicles
IQR	Interquartile Range
KB	Kilo Bytes
ML	Machine Learning
oneVFC	Vehicular Fog Computation
ONNX	Open Neural Network Exchange
SIMD	Single Instruction, Multiple Data
SQRT	Square Root
SW	Software
TF	Tensor Flow
TFLite	Tensor Flow Lite
TPU	Tensor Processing Unit
TVM	Tensor Virtual Machine
VIPS	Valid Images Per Second
VOPS	Valid FLOPs Per Second

Chapter 1

Introduction

In the rapidly evolving field of mobility services, ensuring passenger safety and comfort through advanced technology has become essential [27]. This dissertation focuses on the critical role that the Internet of Things (IoT) plays in improving these characteristics, with a particular focus on implementing machine learning (ML) models in a heterogeneous hardware environment [29]. The main objective is to identify and evaluate ML frameworks that can accelerate the transition from development to production, support the flexible development of ML pipelines, ensure compatibility with a wide range of hardware, including ARM, x86 and GPUs, and maintain computational efficiency, stability and security.

The application of IoT and ML in vehicles allows for better data collection and analysis, providing valuable insights into vehicle behavior and driving conditions. This not only contributes to passenger safety by identifying potential collisions and sudden stops, but also to comfort by adjusting vehicle behavior in real time to offer a smoother travel experience.

However, the transition of ML models from development to production faces significant challenges. The different hardware architectures, such as ARM, x86 and GPUs, require ML frameworks that are flexible and compatible, as well as efficient in terms of computation and security. Adapting these models to resource-constrained environments, such as devices with low computing power, is a critical area in need of innovative solutions.

1.1 Motivation

The motivation behind this work stems from the increasing demand for advanced safety features, personalized comfort, and overall enhancement of the passenger experience in mobility services [27]. As the market grows more competitive, and as consumer expectations rise alongside technological advancements, the need to efficiently deploy sophisticated machine learning models in diverse hardware environments becomes more acute [9]. This deployment is not without its challenges, particularly in ensuring compatibility with various hardware platforms (such as ARM, x86, and GPUs), maintaining computational efficiency, and upholding robust security and stability standards.

> "...Connected vehicles and vehicular networks have been identified as a key technology for enhancing road safety and transport efficiency. In recent years, vehicular network infrastructure integration technology attracts a great amount of attentions; it also brings inestimable economic value, and will play an important role in the next generation of intelligent transportation systems and communication network development." [8, Section: B Deployment Of Typical Applications]

This research not only aims to contribute valuable insights for ML/SW engineers but also to pave the way for more sophisticated, efficient, and user-centered mobility solutions in the future. The preliminary approach involves using data from Inertial Measurement Units (IMUs), including accelerometers, gyroscopes and microphones, to train and evaluate ML models. This data is essential for understanding and improving passenger safety and comfort, representing a practical challenge in applying AI to mobility services.

1.2 Objectives

The specific objective of this project is to identify and evaluate machine learning frameworks that can facilitate the efficient deployment of these models in heterogeneous hardware environments typically found in IoT applications within the mobility sector. By focusing on frameworks such as TensorFlow Lite, ONNXRuntime, and PyTorch, this research aims to determine the best practices for deploying ML models that can operate effectively on resource-constrained devices. Additionally, the project seeks to address the complexities involved in translating models between different frameworks, ensuring compatibility, and maintaining high performance across various hardware configurations.

This dissertation aims to develop solutions that bridge the gap between low-level hardware optimizations and high-level ML model development, ultimately contributing to the creation of reliable and innovative intelligent mobility solutions. Through this research, the goal is to provide insights that will assist in making informed choices when selecting ML frameworks, ensuring efficient, scalable, and robust AIoT solutions for enhancing passenger safety and comfort in the mobility sector.

1.3 Structure

In addition to the introduction, this dissertation consists of five more chapters. Chapter 2 discusses the state of the art and presents related work, focusing on the influence of hardware on models and frameworks, and providing an overview of the different machine learning frameworks considered. Chapter 3 explores the methodology and tools used, including a detailed analysis of machine learning frameworks and the benchmarking process. Chapter 4 presents the results, including a detailed study of the operations and models, as well as an analysis of the Bosch collision detection model. Chapter 5 provides a comprehensive discussion of the benchmarking results, focusing

on execution times, memory usage, and the overall performance of the frameworks in resource-constrained environments. Finally, Chapter 6 concludes the dissertation by summarizing the main findings and outlining future work and possible research contributions. This structure ensures a logical flow of information, guiding the reader from the foundational aspects to the detailed analysis and final conclusions.

1.4 Internship Timeline

During my internship at Bosch, which began on February 5th, I engaged in a series of structured and intensive activities aimed at addressing the problem presented by the company and fulfilling the requirements of this dissertation. Throughout the internship, I participated in weekly meetings with my supervisors at Bosch, ensuring consistent guidance and feedback. This subsection outlines the key phases and milestones achieved during the internship.

- **February: Initiation and Problem Understanding**

  The internship started with an initial phase dedicated to understanding the problem defined by Bosch. This phase involved a thorough analysis of the problem, setting the stage for the subsequent research and development efforts.

- **March: Framework Research and Initial Selection**

  In early March, the focus shifted to conducting an extensive research on the various machine learning frameworks available in the market. This research aimed to identify frameworks that could potentially address the specific needs of the project. The findings from this research led to the first selection of frameworks, as detailed in Section 2.2 of this dissertation.

- **April: Device Analysis and Cross-Compilation Approach**

  The next phase, starting in early April, involved a detailed examination of the hardware device provided by Bosch, described in Section 4.1. This phase included exploring the requirements for cross-compiling the selected frameworks and associated models for the ARMv7 architecture. Encountering several challenges during this process, a second selection of frameworks was made, focusing on those most compatible with the hardware, as discussed in Section 4.2.

- **May: Model Development and Adaptability Testing**

  In May, the focus was on developing various models to evaluate the adaptability of the chosen frameworks. This phase involved creating models specifically designed to test different aspects of the frameworks, as outlined in Section 4.4. The models were then used to select benchmarks that would provide the most relevant insights into framework performance, as

elaborated in Section 2.3.

- **June: Benchmarking and Results Analysis**

  During June, the benchmarks were executed, and the results were analyzed. This phase highlighted compatibility issues, which required adjustments in the analysis approach. By the end of June, I had received the collision detection model from Bosch. The final steps involved cross-compiling this model and obtaining performance results.

The writing of this dissertation was an ongoing process throughout these months. I gradually documented the progress, findings, and analyses, ensuring a comprehensive and coherent presentation of the research and results.

This structured and supervised approach during the internship at Bosch provided a solid foundation for the research and enabled a systematic evaluation of the machine learning frameworks, culminating in the findings presented in this dissertation.

Chapter 2

Background and Fundamental Aspects

2.1 Influence of Hardware on Models and Frameworks

In the context of this project, which uses a specific device with limited hardware, the choice of hardware directly influences the selection of machine learning frameworks and models as follows:

- **Performance and Energy Efficiency:**

  Devices with limited computing resources, such as the ARMv7, require frameworks optimised for energy efficiency and performance. The hardware's ability to process inferences quickly affects the overall efficiency of the system, especially in IoT applications where latency and response time are critical. Machine learning models must be able to operate efficiently without overloading the device, guaranteeing minimal energy consumption.

- **Model size:**

  The memory and storage capacity of the hardware influences the size and complexity of the models that can be implemented. Larger models may need to be quantised or simplified to fit the limitations of the device. Quantisation techniques help to reduce model size without significantly sacrificing accuracy, allowing models to work within hardware constraints.

- **Latency and Response Time:**

  In IoT applications, latency and response time are critical factors. The ability of the hardware to process inferences quickly affects the overall efficiency of the system. Frameworks optimised for mobile and embedded devices are designed to provide fast response times, essential for applications that require real-time processing.

- **Ease of Implementation and Integration:** Hardware compatibility with different machine learning frameworks can simplify or complicate implementation. Choosing frameworks that

natively support the available hardware facilitates integration and reduces the need for complex adaptations. For example, using frameworks that offer direct support for hardware-specific instructions, such as SIMD (Single Instruction, Multiple Data) and floating point units, allows for more efficient execution of models.

These factors highlight the importance of considering the capabilities and limitations of the hardware when selecting frameworks and developing machine learning models to ensure optimal and efficient performance.

2.2 Frameworks

Before we delve into the identification and evaluation of machine learning frameworks, it is important to understand what a framework is. A machine learning framework is a platform that provides tools, libraries and interfaces that facilitate the development, training, validation and deployment of machine learning models. These frameworks are essential because they abstract away the complexity of the underlying mathematical operations and offer a structured environment for developing machine learning algorithms more efficiently and effectively. They allow developers to focus more on creating innovative models and less on detailed technical implementation. Popular examples of frameworks include TensorFlow, PyTorch and ONNX Runtime, each with its own specific characteristics and advantages.

The central problem of this dissertation is to identify and evaluate suitable machine learning frameworks such as TensorFlow, TensorFlow Lite, TFLite Micro, PyTorch, Pytorch Mobile, ONNX Runtime and Apache TVM. These frameworks should facilitate the rapid transition of AI models from development to deployment in environments with low computing power. Solving this problem has significant implications: it could lead to the development of more advanced, reliable and user-friendly mobility services, substantially improve passenger safety and comfort, and foster innovation in a sector that is increasingly dependent on cutting-edge AI and IoT.

2.2.1 TensorFlow

TensorFlow is an open source framework developed by Google that stands out in the field of machine learning due to its flexibility, scalability and robustness. It is widely used in both academic research and industrial applications, enabling the creation, training and implementation of machine learning models and deep neural networks.

For this dissertation, TensorFlow is extremely important, as it will be one of the main frameworks used throughout development. Its extensive library of tools and APIs makes it easy to build complex models and perform advanced machine learning tasks. In addition, TensorFlow supports a wide range of devices, from CPUs and GPUs to TPUs, providing a versatile and efficient execution environment.

However, for this specific project, TensorFlow Lite (TFLite), which will be analyzed later, is more suitable.

TensorFlow will, however, be used mainly for the initial development of the models. After training and validation, these models will be converted to the TensorFlow Lite format, ensuring that they can be run efficiently on the hardware. In addition, TensorFlow allows export to other frameworks, such as ONNX, increasing flexibility in the implementation process.

2.2.2 TensorFlow Lite

Among the machine learning frameworks considered in this dissertation, TensorFlow Lite emerges as a particularly significant player. TensorFlow Lite is an open-source deep learning framework developed by Google, designed specifically for on-device inference with low latency and a small binary size. It stands out in the realm of AIoT applications due to its ability to bring high-performance machine learning models to edge devices, including those with limited computational power [14].

TensorFlow Lite's optimization for mobile and embedded devices makes it an ideal choice for mobility services, where the need for real-time processing and responsiveness is paramount. Its capability to efficiently run ML models on a wide range of hardware platforms, from high-end smartphones to resource-constrained microcontrollers, aligns perfectly with the diverse hardware requirements in modern mobility solutions. This includes compatibility with ARM, x86 architectures, and support for GPU acceleration, ensuring versatile deployment across different device types [14].

2.2.3 TensorFlow Lite for Microcontrollers

In addition to TensorFlow Lite, this dissertation also delves into TensorFlow Lite for Microcontrollers (TFLite Micro), an extension of TensorFlow Lite tailored for extremely resource-constrained environments. TFLite Micro is specifically designed to run machine learning models on microcontrollers and other devices with only a few kilobytes of memory. Its development reflects the growing trend of deploying AI capabilities directly onto the smallest of edge devices, a crucial aspect in the burgeoning field of AIoT [11].

One of the key strengths of TFLite Micro lies in its minimalistic design. It requires no operating system support, no dynamic memory allocation, and has an exceptionally small binary size. This makes it suitable for running in the constrained environments typical of microcontrollers found in many IoT devices [11].

By incorporating TFLite Micro into the research scope, this dissertation acknowledges and addresses the growing need for lightweight, efficient, and reliable machine learning solutions that can operate on the very edge of the network. The framework's focus on microcontroller compatibility, coupled with its ability to run with limited resources, presents a compelling option for the future of intelligent mobility services.

2.2.4 PyTorch Mobile

PyTorch Mobile was developed to allow Machine Learning models to be run on mobile devices, taking advantage of PyTorch's flexibility and ease of use.It offers optimized support for simpler hardware environments, which initially seemed to make this framework a potentially suitable choice for the 32-bit ARMv7 used in the project. PyTorch Mobile allows developers to convert their PyTorch models to an optimized version that can be deployed on mobile devices with limited resources, offering a balance between performance and efficiency. [22].

However, PyTorch Mobile's compatibility is predominantly focused on Android and iOS platforms. This orientation towards the most popular mobile operating systems results in a lack of direct support and optimization for other types of hardware, such as the 32-bit ARMv7 which does not fall into the typical categories of mobile devices supported by the framework.This limitation means that, despite the potential advantages of PyTorch Mobile in terms of running models on hardware with limited resources, it is not a viable choice for the specific hardware used in this project. The lack of compatibility and support for ARMv7 outside the mobile device ecosystem makes it impossible to use PyTorch Mobile efficiently in this context.

Therefore, although PyTorch Mobile offers interesting features for developing Machine Learning applications on mobile devices, its applicability in this project is limited due to its restricted compatibility with Android and iOS platforms. This reinforces the need to consider other frameworks that offer more direct and optimized support for 32-bit ARMv7 hardware.

2.2.5 ONNX Runtime

Another pivotal framework under consideration in this dissertation is ONNX Runtime, an open-source performance-focused inference engine for ONNX (Open Neural Network Exchange) models. Developed by Microsoft, ONNX Runtime is designed to optimize model performance across a wide range of platforms and devices, making it highly relevant for AIoT applications in mobility services [2].

ONNX Runtime stands out due to its versatility and cross-platform compatibility. It supports a variety of hardware configurations, including ARM and x86 architectures, and can efficiently operate in both CPU and GPU environments [2]. This adaptability makes it an excellent choice for deploying machine learning models in diverse hardware settings encountered in the mobility sector, from high-powered servers to edge devices.

Incorporating ONNX Runtime into this research provides an opportunity to explore how a framework can bridge the gap between different ML development environments and the varied hardware ecosystems in the mobility services sector. It highlights the framework's potential to enhance the portability and performance of machine learning models, ensuring they are not only accurate but also efficient and adaptable to the dynamic needs of modern transportation solutions.

2.2.6 Apache TVM

Apache TVM is an open-source machine learning compiler framework that transforms machine learning models into deployment-ready code for a variety of hardware platforms [1]. Its key strength lies in its ability to automatically optimize models for specific target hardware, making it a valuable tool for AIoT applications where efficiency across diverse devices is crucial [2]. Although it plays a less central role in this dissertation compared to other frameworks, Apache TVM's unique approach to model optimization and its flexibility in supporting multiple front-end frameworks like TensorFlow and PyTorch make it an interesting candidate for certain AIoT applications in the mobility sector.

2.3 Benchmarking Machine Learning Frameworks

A fundamental aspect of this research is the benchmarking of selected machine learning frameworks, such as TensorFlow Lite, PyTorch Mobile, ONNX Runtime, among others. This benchmark analysis will be crucial in determining the suitability of each framework for the specific needs of mobility services, considering the ARM Cortex-A7 32-bit hardware environment.

The main benchmarking criteria include:

- **Processing Speed** — Evaluate how quickly each framework processes data, a critical factor for real-time applications in mobility services;
- **Memory Utilisation** — Measuring the memory space required for each framework to work, an important aspect in devices with memory restrictions;
- **Ease of Integration and Use** — Evaluate how easily each framework can be integrated into existing systems and how simple it is to use;
- **Flexibility and Scalability** — Examine how the frameworks adapt to different sizes and types of data, as well as their ability to scale;

These benchmarks will provide valuable insights not only into technical performance, but also into the practical viability of frameworks in mobility applications. Understanding these aspects is crucial when choosing a framework.

2.4 Understanding Artificial Intelligence of Things(AIoT)

The Artificial Intelligence of Things (AIoT) is an emerging field that combines the data collection capabilities of the Internet of Things (IoT) with the decision-making capabilities of Artificial Intelligence (AI). AIoT aims to make IoT devices smarter by enabling them to act on the basis of the data they collect in real time. AIoT applications span several domains, including smart homes, healthcare, industrial automation and mobility services. In the context of mobility services, AIoT can transform the way vehicles and traffic systems interact with the environment, leading to greater

safety, efficiency and better experiences for users. For example, AIoT can enable autonomous vehicles to make real-time decisions based on data from various sensors, improving navigation and safety.

The integration of AI and IoT brings several benefits. Improved decision-making is one of the main advantages, as AI algorithms can process and analyse large amounts of data collected by IoT devices, providing insights and enabling predictive maintenance, anomaly detection and real-time decision-making. In addition, AIoT systems can optimise the use of resources, reduce energy consumption and increase operational efficiency in various applications. This improved efficiency can lead to cost savings and more sustainable operations. In addition, AIoT can offer personalised and adaptive services, increasing user satisfaction and engagement. By adapting responses and actions based on real-time data and learnt patterns, AIoT systems can provide more relevant and timely interactions with users.

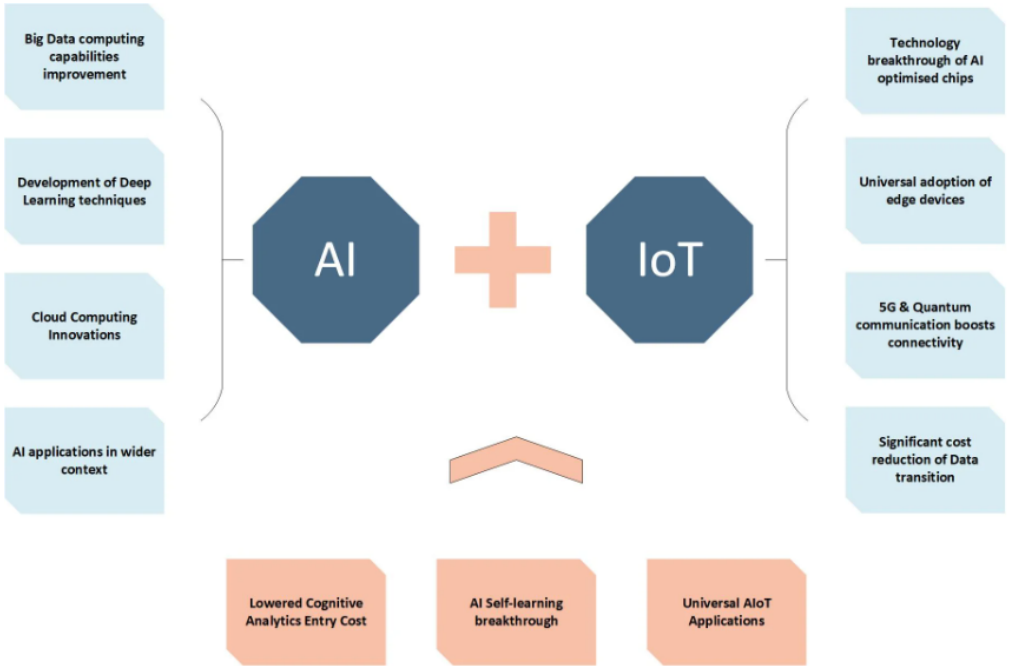


Figure 2.1: AIoT [7].

2.5 Machine Learning Frameworks in Artificial Intelligence of Things

In AIoT, a 'framework' typically refers to a comprehensive suite of tools and libraries designed for developing and deploying machine learning models. These frameworks provide essential building blocks for creating complex algorithms and simplify the process from model development to deployment, especially in varied hardware environments. Frameworks like TensorFlow Lite and PyTorch Mobile are specifically tailored for deploying machine learning models on edge devices, which are a central component of AIoT in mobility services.

2.6 What is a Machine Learning Model?

For the purposes of this dissertation, understanding what constitutes a machine learning model is fundamental. A machine learning model is a file created by training an algorithm on data to recognize patterns and make decisions or predictions. These models are key in mobility services for tasks like predicting traffic patterns or enabling autonomous vehicle functions. The dissertation will delve into how these models can be effectively developed, optimized, and deployed in AIoT environments.

2.7 Implementation and Deployment Challenges of Machine Learning Models

Deployment refers to the process of implementing a trained machine learning model in real-world applications, a critical step in AIoT.

The process of deploying a machine learning model begins with a clear problem definition, where the objectives and requirements of the ML solution are established. This is followed by the crucial stage of data collection and preparation, involving gathering, cleaning, and preprocessing data to make it suitable for machine learning. Next is the model selection and development phase, where an appropriate ML algorithm is chosen based on the problem type and relevant features are engineered for training. The model is then trained using the prepared dataset, and its performance is validated using a separate dataset to ensure it generalizes well to new data. This step is critical to prevent overfitting and optimize the model's effectiveness. Model evaluation follows, using a test dataset and relevant performance metrics to assess how well the model will perform in real-life scenarios. After the model has been optimized through hyperparameter tuning [1] and potentially simplified for efficiency, it is ready for deployment. This involves preparing the deployment environment (which could be a server, cloud platform, or edge device), integrating the model into the existing infrastructure or application, and setting up continuous monitoring systems. The following Figure 2.2 helps to understand a bit more about this concept:

[1] Hyperparameter tuning is a critical step in the machine learning model development process. It involves adjusting the parameters of a machine learning algorithm to optimize the performance of the model.

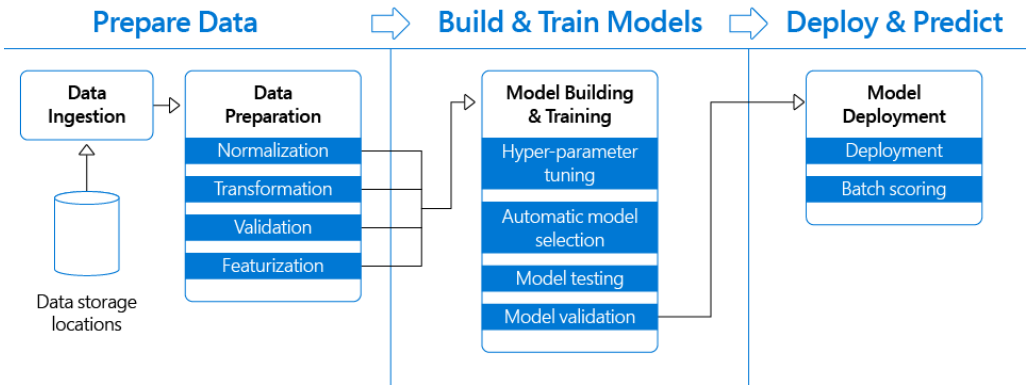


Figure 2.2: Overview of the Machine Learning Workflow [5].

The effective deployment of ML models in AIoT requires careful consideration of several factors. In addition to model and framework selection, it is crucial to consider the execution environment. ML models need to be optimised to run efficiently on devices with limited resources, such as microcontrollers. This involves not only reducing the size of the model, but also ensuring that the model maintains high prediction accuracy. In addition, the deployment process must include a robust monitoring and maintenance system to ensure that the model continues to operate efficiently and accurately over time, adapting to changes in data patterns and the operating conditions of the device.

2.8 Model Optimization Techniques

Model optimisation techniques are essential for deploying machine learning models on devices with limited resources, such as IoT devices, mobile phones and embedded systems. These techniques help to reduce model size, decrease inference and execution time and improve energy efficiency without significantly compromising accuracy.

In the context of this project, quantisation was a key technique used to optimise machine learning models for deployment on devices with limited resources. Post-training quantisation was applied to convert 32-bit floating point model weights to 8-bit integers, resulting in a significant reduction in model size and an improvement in computational efficiency. This technique allowed the models to maintain acceptable accuracy while operating within the memory and processing constraints of the target devices.

In addition, training-aware quantisation techniques were explored, where quantisation is simulated during model training. This helped ensure that the model remained robust to the inaccuracies introduced by quantisation, resulting in better final accuracy compared to quantisation applied after training. These techniques have been crucial in achieving a balance between performance and accuracy, enabling the effective implementation of machine learning models in IoT environments and embedded systems.

2.9 Inertial Measurement Units

IMU data from Inertial Measurement Units is crucial for a wide range of applications in mobility services, especially in the context of AIoT. This data is collected by IMU sensors, which generally include accelerometers, gyroscopes and, in some cases, magnetometers however these will not be used in this work [30].

- **Accelerometers** — Measure linear acceleration in one or more directions. In the context of vehicles, they can be used to detect changes in speed or to monitor driving behaviour;
- **Gyroscopes** — They measure rotation or rate of spin. They are useful for understanding vehicle orientation and are essential in navigation systems.

These sensors together provide a comprehensive view of the movement and orientation of a device or vehicle and will be used to feed the ML algorithms.

Chapter 3

Bibliographical Review and Related Work

Continuing our understanding of AIoT and machine learning frameworks, this section sets out to explore the latest developments in these domains. Here, it will be presented a detailed overview of current research, the advances made and the gaps that still exist.

3.1 Current State of Artificial Intelligence of Things in Mobility Services

To identify the current state of AIoT's contribution to mobility services, an analysis of several studies reveals significant advances and innovative approaches. Examples of these studies could be the following [6] and [20] which explore everything from IoT-based vehicle monitoring systems to advanced vehicle fog computing platforms, highlighting the potential of AIoT to transform the efficiency, safety and user experience of mobility services.

In the study [6] presents a low-cost solution for acquiring data from vehicle Electronic Control Units (ECUs) during road tests. The proposal integrates affordable acquisition hardware with an IoT server, allowing data to be collected and transmitted in near real time. The solution was compared to commercial systems, demonstrating similar performance in terms of data acquisition rate, but at a significantly lower cost. The study also highlighted the integration of AI algorithms for data processing, emphasising the efficiency, innovation and feasibility of the approach. This work offers important insights into the potential of AIoT to improve efficiency and reduce costs in vehicle testing, contributing to the intelligent development of mobility services.

The study [20] proposes a vehicular fog computation platform (oneVFC) for AI applications in the Internet of Vehicles (IoV). This platform uses the oneM2M standardisation for interoperability and hierarchical organisation of resources. oneVFC manages distributed resources and orchestrates information flows and computing tasks in vehicular fog nodes, demonstrating efficiency in reducing application processing time, especially in situations of high workload or frequent requests. The research also explores the training of AI models using the Federated Learning

approach, preserving privacy and saving communication. This study complements research into AIoT in mobility services, highlighting the effectiveness of distributed computing and intelligent resource management in IoV environments.

Both of this studies represent notable contributions to the development of intelligent mobility solutions, reflecting the state and the growing integration of advanced technologies in the field of mobility.

3.2 Machine Learning Frameworks in Diverse Hardware Environments

This section is dedicated to analyzing the behavior of different Machine Learning frameworks in various hardware environments. One of the most significant studies for this dissertation is [16], which focuses on evaluating the performance of frameworks such as TensorFlow Lite and PyTorch Mobile on mobile devices. This study is essential for understanding the differences in efficiency and compatibility between the frameworks, vital information for your research which also uses TensorFlow Lite and PyTorch Mobile in environments with limited computing capacity.

Although neural networks are not used in this dissertation and the focus is on mobility services, the study [16] is important. It provides insights into computational efficiency and memory usage on different devices, crucial aspects for any ML framework, including those used in systems not based on neural networks. In addition, the study helps to understand how different frameworks can be adapted or optimized for specific environments, which is key to implementing efficient AIoT solutions in mobility services.

The results of the study offer valuable insights into the compatibility of frameworks with various hardware configurations, a central concern of his research. For example, significant variations in performance and accuracy were observed between different combinations of models and frameworks on specific devices. These findings are crucial to this dissertation, which seeks to identify frameworks suitable for a wide range of hardware, including devices with limited computing power such as the ARM Cortex-A7.

Although accuracy and processing times vary between devices and frameworks, the study indicates that when comparing TensorFlow Lite and PyTorch Mobile, TensorFlow Lite generally shows better inference and model loading times on all devices and models tested. However, it is important to note that, in some cases, this may come at the cost of slightly lower accuracy. This finding is particularly relevant to this dissertation, which also focuses on these two frameworks, providing a crucial reference point for the performance analysis that will be carried out.

In addition, the study proposes unified metrics for evaluating the AI capacity of devices, such as Valid Images Per Second (VIPS) and Valid FLOPs Per Second (VOPS). Although Valid Images Per Second (VIPS) is not important for this thesis, since it is a metric used to evaluate a system's ability to process images in an AI context, which is not important in this context. On the other hand, VOPS (Valid FLOPs Per Second) is a crucial metric for your dissertation when benchmarking ML frameworks.

FLOPs (Floating Point Operations Per Second) is a measure that indicates how many floating point operations a processor or system can perform per second and is fundamental to understanding the processing capacity of a device. In this research, which focuses on processing efficiency and speed, FLOPS are essential for comparing the computational performance of different frameworks. They help determine how quickly and efficiently data can be processed, a key factor in the effective implementation of AIoT solutions in mobility services. Although the use of FLOPs is very important for the analysis of computational performance, the study [21] brings a new perspective to benchmarking in AIoT, highlighting that, despite the importance of using FLOPs as a performance indicator, there are significant limitations to this approach, especially for mobile GPUs.

The research proposes alternative methods, such as the use of gradient boosting, for latency prediction. While the FLOPs-based approach provides a simple and straightforward measure, it may not capture the complexity of inference latency on mobile devices. On the other hand, methods such as gradient boosting allow for more detailed analysis and can lead to more accurate predictions, although they require more computational resources. Reading this paper helped me realise that there are other methods for benchmarking the computational capacity of each framework.

This brings us to a negative point of the reseach [16], the lack of analysis of the energy consumption of each framework. This aspect is crucial, especially in mobile devices where energy efficiency is paramount. In addition, the study may not have explored the full range of hardware environments that are relevant to AIoT applications, such as systems with extremely limited resources. These limitations are important to consider when applying the study's findings to this dissertation.

But the study [21] faced some limitations too. One of these is the accuracy of the proposed methodology, which may not be as precise as direct measurements. This is partly due to the training sample used, which may not adequately represent the entire search space of neural networks. In addition, the approach used was not successful in all cases, suggesting limitations in the method's effectiveness in certain situations. Another notable limitation is that direct measurements may require a large number of runs to obtain consistent results, indicating that the process of accurate measurement can be resource-intensive and time-consuming. In addition, the lookup table method, effective on CPU-based devices, did not show the same accuracy on GPUs, indicating limitations in the applicability of certain latency prediction methods on different types of devices

3.3 Cross-Compilation and Implementation of Models on ARM Devices

Implementing machine learning models on ARM devices requires a meticulous cross-compilation process to ensure that the trained models are compatible and optimised for the target hardware.

This process involves several critical steps, from converting the models to suitable formats to applying optimisation techniques specific to the target hardware.

ARM devices, which are widely used in IoT applications and mobile devices, present unique challenges due to their memory and computing power limitations. The article [31] provides a comprehensive overview of the potential of artificial intelligence on edge devices, highlighting the importance of optimisation methods to adapt neural network models to the constraints of these devices. To begin the cross-compilation process, models trained in development environments, usually using frameworks such as TensorFlow or PyTorch, need to be converted to a format compatible with ARM devices. This conversion often uses TensorFlow Lite, a simplified and optimised version of TensorFlow designed specifically for mobile and embedded devices. TensorFlow Lite allows complex models to be converted into compact and efficient versions, ready to run on devices with limited resources [31].

The article [12] on deep compression illustrates how pruning and quantisation techniques can significantly reduce the size of models without loss of precision, and these techniques are crucial for efficient implementation on ARM devices. Pruning removes redundant or insignificantly weighted connections, while quantisation reduces the precision of model weights, reducing the need for storage and increasing the speed of inference.

Quantization is a crucial technique in this process, significantly reducing the size of models and improving inference performance on ARM devices. The paper [4] provides an elaborate performance characterization of quantization techniques, such as FP16 and INT8, on edge devices using frameworks like TensorFlow Lite, ONNX, and PyTorch. The study demonstrated that quantized models, particularly those using INT8, deliver substantial performance improvements without compromising accuracy. For example, INT8-based quantized models delivered 4× better performance over FP32 using TensorFlow Lite on a Raspberry Pi device.

Moreover, cross-compilation involves using various tools and libraries to optimize models for ARM architectures. The use of the Arm NN library, which bridges the gap between neural network frameworks and ARM hardware, is one such example. This library enhances the performance of models by optimizing them for ARM Cortex-A CPUs.

The deployment process also includes the use of post-training quantization techniques, where models are quantized after they are fully trained. This approach, as detailed in the paper [4], involves using representative datasets for calibration to ensure that the quantized models maintain high accuracy while benefiting from reduced size and increased inference and execution speed.

In conclusion, cross-compilation and implementation of machine learning models on ARM devices require a combination of model conversion, quantization, and optimization techniques. By leveraging tools such as TensorFlow Lite and Arm NN, and employing methods like post-training quantization, developers can effectively deploy efficient and accurate models on resource-constrained ARM devices.

3.4 Positioning Current Work in the State of the Art

This research firmly positions itself within the evolving landscape of AIoT in mobility services through a systematic evaluation of machine learning frameworks for deployment on heterogeneous hardware platforms. By meticulously selecting and evaluating frameworks such as TensorFlow Lite and ONNX Runtime, and conducting an in-depth analysis of contemporary studies, this work addresses critical challenges related to processing efficiency and hardware adaptability. The benchmarks designed for this dissertation, centered on 32-bit ARM Cortex-A7 processors, aim to expand the capabilities within the constrained computing environments typical of AIoT devices.

The implementation of cross-compilation and optimization strategies for ARM devices further positions this work within the state of the art. Cross-compilation is an essential process for adapting machine learning models to run on ARM architectures, which are prevalent in IoT applications. This process involves converting models trained in environments like TensorFlow or PyTorch into formats compatible with ARM devices using tools such as TensorFlow Lite, ONNX Runtime, and Arm NN. Optimization techniques such as quantization and pruning ensure that the models operate efficiently within the constraints of ARM hardware. Quantization, particularly post-training quantization, has demonstrated significant reductions in model size and improvements in inference performance on ARM devices without compromising accuracy. Studies cited in this dissertation show that INT8 quantization can deliver up to four times better performance compared to FP32 on devices like the Raspberry Pi.

Although LightGBM was not used in this dissertation, studying this gradient boosting framework helped in understanding different types of machine learning models, such as the one provided by Bosch. LightGBM is known for its efficiency and speed, particularly in handling large datasets and complex models, and this understanding was beneficial in contextualizing the machine learning landscape relevant to the project's goals.

By aligning this study with the rigorous demands of real-world mobility scenarios, it contributes to the advancement of AIoT applications in the field of transport and smart mobility. This research offers a comprehensive framework for future explorations into the seamless integration of AIoT solutions into mobility services. A comprehensive benchmarking study could significantly enhance the process of selecting deployment frameworks, ensuring that the chosen technology is both efficient and adaptable to the targeted environment.

In addition to the studies previously discussed, the article [13] provides a comprehensive examination of the capabilities of modern 32-bit microcontrollers (MCUs) in IoT and Big Data contexts. The research highlights the evolution of MCU performance, particularly focusing on their ability to handle complex data processing tasks locally, thereby alleviating the burden on network resources and central processing units.

This study underscores the significant advancements in MCU technology, particularly with the integration of Floating Point Units (FPU) and Digital Signal Processing (DSP) capabilities. These enhancements enable MCUs to perform sophisticated data preprocessing tasks, which are crucial for IoT applications that require real-time data analysis and decision-making. The paper

also discusses the implications of these advancements for the overall efficiency and scalability of IoT systems, emphasizing how local processing capabilities can lead to more efficient use of network and Big Data resources.

Moreover, the article provides valuable insights into the performance benchmarks of various MCU architectures, such as ARM Cortex-M and Tensilica Xtensa LX106, illustrating their capacity to support high-speed data transmission and complex computational tasks. These benchmarks are critical for evaluating the suitability of different MCUs for specific IoT applications, particularly those involving high volumes of sensor data and requiring robust real-time processing capabilities.

The findings of the study [13] align with the objectives of this dissertation, which aims to identify and evaluate machine learning frameworks that can efficiently operate in resource-constrained environments. The detailed performance analysis of MCUs presented in their study complements the framework evaluation discussed in this dissertation, providing a broader context for understanding the hardware capabilities that underpin effective ML model deployment in IoT systems.

Chapter 4

Model Deployment Flows for Embedded Platforms

Chapters 2 and 3 focused on the background knowledge required for this work and the current state of this knowledge, i.e., the available implementations. This chapter will examine the procedures, techniques, and difficulties associated with translating Machine Learning models and various operations (such as SORT and FFT) from Python to C++ for deployment on an ARMv7 32-bit processor.

The purpose of this Chapter, as explained in Chapter 1, is to optimize the translation process to obtain an efficient implementation capable of running on the limited ARMv7 32-bit hardware. The different frameworks used, the difficulties encountered, and the solutions developed during the process will be discussed, focusing on the practical application of the models and operations to the use case of this project. The research aims to find the best way to carry out this translation using different frameworks, guaranteeing the effectiveness and efficiency of the models under actual conditions of use.

In addition, this section will discuss the methodology used to study the different frameworks, explaining the various operations that will be analyzed. The aim is to evaluate the compatibility and efficiency of each "flow" (the path used to deploy the model on ARM), ensuring that the solutions developed can later be applied to the project's final model. This evaluation will include an analysis of the compatibility of each framework with the hardware and operation in question, as well as the effectiveness of the optimization techniques applied.

4.1 Detailed Specifications of Deployment Hardware

First, it is necessary to specify the hardware used in the deployment and testing of the models in this project. The device used is the Qualcomm MDM9207, which includes the 32-bit ARMv7 processor, known for its energy efficiency and suitability for IoT applications.

The ARMv7 processor is a single-core that operates at a frequency of up to 1.5 GHz, balancing performance and power consumption. This processor is equipped with several advanced

features crucial for the efficient processing of Machine Learning models. These include support for half-precision floating-point instructions, the VFP (Vector Floating Point) unit, and NEON SIMD (Single Instruction, Multiple Data) technology, which efficiently enables multimedia and signal processing operations.

In addition, the ARMv7 includes the third and fourth versions of the vector floating point unit (VFPv3 and VFPv4), with VFPv4 supporting double precision floating point operations, which is essential for more precise and complex calculations. These features make ARMv7 a robust choice for running deep learning models on devices with limited resources, ensuring efficient and reliable performance. Understanding these specifications will be fundamental for the next sections, where the compatibility and optimization of the cross-compilation process for this specific hardware will be discussed.

4.2 Analysis and Evaluation of Frameworks

At the start of the project, comprehensive research was carried out on the different machine learning frameworks available. The focus of this research was to understand the advantages and disadvantages of each framework, especially in terms of compatibility with the specific hardware used in this project. This analysis was crucial in guiding the development process and selecting the most appropriate frameworks to address the specific challenges related to AIoT in mobility services [17].

In this subsection, the frameworks considered and analyzed will be explained, detailing which were actually implemented and which were not, including the reasons why some were not viable. The frameworks studied include TensorFlow Lite, TensorFlow Lite Micro, ONNX Runtime, PyTorch, and Apache TVM [26, 14, 22, 1, 2].

The main objective here is to provide a clear overview of how each framework was evaluated in terms of compatibility with the 32-bit ARMv7 hardware used in the development of this project. Frameworks that did not meet the compatibility criteria were discarded, and this subsection will explain these decisions in detail, setting the stage for more in-depth discussions in the following subsections about each of these frameworks.

Below is an overview of the paths (“flows”) studied for developing and deploying Machine Learning models in the context of this project.

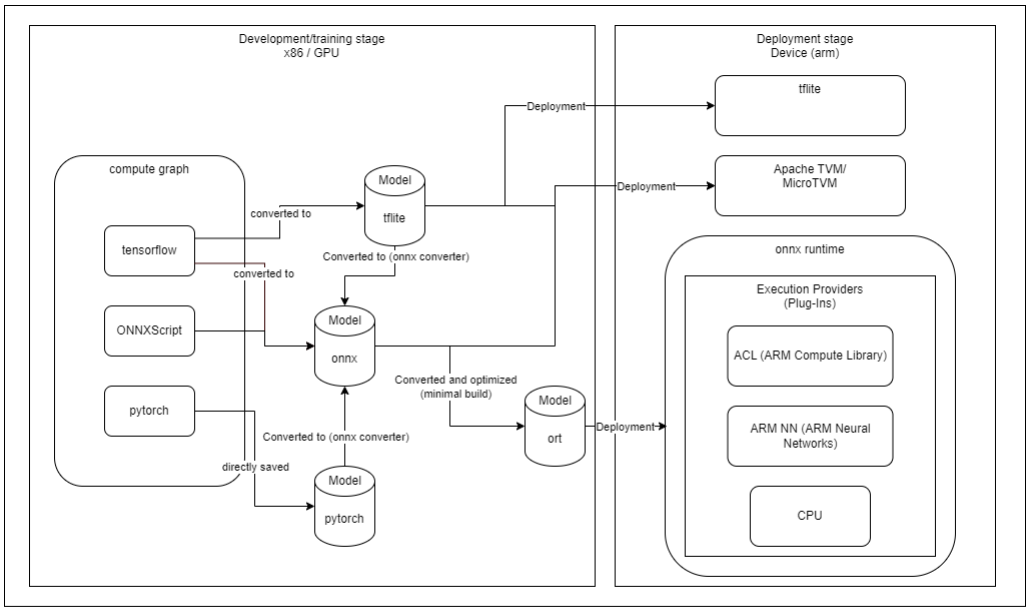


Figure 4.1: Initial framework Interoperability diagram for deployment.

Figure 4.1 shows a diagram illustrating the various paths that Machine Learning models can follow from the development phase to deployment on the ARMv7 device. These paths, or "flows", are made up of different frameworks and conversion methods that allow the models to be adapted to the specific hardware environment used in this project.

4.2.1 TensorFlow Lite Avaluation

For several reasons, TensorFlow Lite (TFLite) is an excellent choice for implementing Machine Learning models on the 32-bit ARMv7 processor. Firstly, TFLite has been specifically designed to enable model inference on mobile and embedded devices, where processing and power resources are limited. This feature is crucial for running on a 32-bit ARMv7 processor, which is common in IoT devices and embedded systems. TFLite operates with low latency and has a reduced binary size, ensuring that models can be run efficiently even in environments with severe resource constraints. In addition, TFLite is highly compatible with the ARM architecture, including 32-bit ARMv7. This compatibility is supported by optimizations that take advantage of the SIMD (Single Instruction, Multiple Data) instructions of the NEON technology present in ARMv7, allowing the processor to perform multiple data operations in parallel, significantly increasing performance for vectorized operations common in model inference.

ARMv7 has several versions of vector floating point units, such as VFPv3 and VFPv4, which TFLite takes advantage of. VFPv4, for example, supports double-precision floating point operations, which are essential for more precise and complex calculations, providing reliable performance for Machine Learning applications. Support for half-precision floating-point instructions

allows for more efficient use of memory and processing, essential on devices with limited resources. TFLite also includes several quantization optimizations that allow models to use fewer bits to represent weights and activations, reducing processing load and memory consumption,. These features are especially useful for ARMv7, where computing power and memory are limited. This framework therefore proved to be viable and worked adequately in the context of this project.

4.2.1.1 TesorFlow Lite FlatBuffer Advantages

To convert a trained TensorFlow model to run on ARMv7 devices, the TensorFlow Lite converter Python API should be used. This will convert the model into a FlatBuffer, reducing the model size, and modifying it to use TensorFlow Lite operations.

FlatBuffer is an efficient serialization system developed by Google, used primarily for inter-process communication or for storing data efficiently. It is similar to other serialization systems like Protocol Buffers but is specifically designed to be faster and more memory-efficient. One of the most notable features of FlatBuffer is its ability to access serialized data without the need for prior parsing or unpacking. This means elements within a FlatBuffer can be accessed directly from the binary buffer, which is extremely fast and efficient in terms of CPU and memory usage.

FlatBuffers are designed to have minimal memory overhead. They do not require the allocation of additional objects in memory to be accessed, making them ideal for systems with limited resources, such as embedded devices. FlatBuffers allow for the evolution of data schemas (the structure defined for stored data). Fields can be added to the schema without breaking code that uses older versions of the schema, which is excellent for developing applications that may need schema updates over time. They are designed to be used across various platforms and programming languages, making them a popular choice for systems that interact between different types of devices or operate in diverse software environments.

4.2.2 TensorFlow Lite for Microcontrollers Evaluation

TensorFlow Lite for Micro-controllers (TFLite Micro) looked like a promising framework due to its similarities to TensorFlow Lite, but with the potential to be even more computationally viable. Designed specifically to run on micro-controllers, TFLite Micro is optimized for extremely resource-limited devices, such as the Arm Cortex-M7. Given its focus on efficiency and small size, this initial feature indicated that it could be an excellent choice for the 32-bit ARMv7.

However, upon further analysis, it was found that TFLite Micro is aimed more at micro-controllers, such as the Cortex-M7, and not at processors, such as the ARMv7. Although TFLite Micro can also take advantage of NEON's SIMD (Single Instruction, Multiple Data) instructions, just like TFLite, and VFP (Vector Floating Point) units such as VFPv3 and VFPv4, its implementation is more simplified and focused on devices with minimal resources. Although it can use some of these capabilities, it doesn't do so as efficiently or comprehensively as TFLite, which is

designed to make the most of these features on more robust devices. This crucial difference limited its applicability, as the ARMv7 has different characteristics and requirements, necessitating more robust support than FLite Micro was designed to provide.

Another significant reason for the unfeasibility of TFLite Micro in the context of this project was its incompatibility with essential operations for the model under development. Specifically, TFLite Micro did not adequately support flow control operations, such as "While" and "If," and data transformation operations, such as "ReduceSum" and "ReduceMax." These operations are fundamental for the efficient and correct execution of the planned Machine Learning models.

Due to these limitations, continuing to develop and study his framework was not feasible. Incompatibility with the specific hardware and the lack of support for crucial operations led to the decision to focus on other frameworks more suited to the 32-bit ARMv7.

4.2.3 Open Neural Network Exchange Evaluation

The Open Neural Network Exchange (ONNX) is a distinctive framework in the Machine Learning ecosystem, mainly due to its ability to serve as a universal intermediary for converting models between different frameworks. This feature was used extensively in this project, where converting models from frameworks such as PyTorch and TensorFlow Lite (TFLite) to the ONNX format was essential. The versatility of ONNX has enabled effective interoperability between different machine learning tools, making it easier to use the models developed in different execution environments.

4.2.3.1 Conversion to ONNX Model

One of the features most used in this project was precisely this conversion capability. Through ONNX, models created in frameworks such as PyTorch and TFLite could be converted to ONNX format, allowing for more efficient adaptation to the requirements of 32-bit ARMv7 hardware. This ability to act as an effective means of conversion between different frameworks made ONNX a crucial component in the project's workflow, guaranteeing flexibility and rapid adaptation to different execution environments.

To convert TensorFlow models to ONNX, two methods were explored: using the `tf2onnx` tool and the `tflite2onnx` tool. After experimenting with both, it was determined that `tf2onnx` was the preferred method. Although no significant differences were observed in the models generated by either method, `tflite2onnx` had many more operations that were not compatible. Specifically, `tflite2onnx` supports about 31 TFLite operations, whereas TFLite itself has 127 built-in operators. This incompatibility issue makes `tf2onnx` a more robust and reliable option for converting TensorFlow models to ONNX (Fig. 4.2).

In addition, although there is a tool under development called ONNXScript [3], which aims to allow the creation of ONNX models directly, without the need for conversion from other frameworks, it was used in some instances in the development of the project, especially when there were operations that the conversion to ONNX could not process. However, due to its complexity of use,

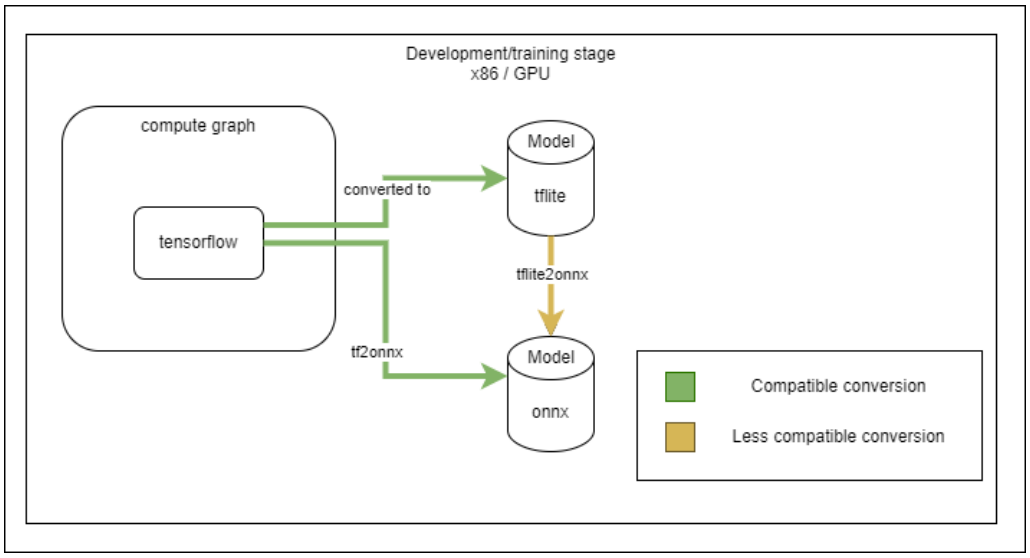


Figure 4.2: TensorFlow to ONNX flow chart.

and because it is still a tool under development, ONNXScript was left aside in favor of more practical and efficient methods. The main focus with ONNX remained on its ability to convert models already existing in other frameworks to the ONNX format, which proved to be more practical and efficient for the purposes of this project.

In the context of this project, using ONNX as a conversion tool proved to be highly viable and efficient. ONNX's flexibility and ability to integrate easily with other tools ensured that it was a relevant and versatile framework, which was subsequently studied and used in the development and deployment phases. This versatility allowed the models to be efficiently adapted to the specific ARMv7 hardware environment, meeting the performance and compatibility requirements necessary for the success of the project.

4.2.3.2 Deployment of ONNX Models

In addition, ONNXRuntime was used to deploy the ONNX models. ONNXRuntime is an inference engine that allows ONNX models to be executed efficiently on a wide variety of hardware, including ARMv7. It offers support for several execution providers, which allow hardware-specific optimizations and ensure that models can be executed with high performance. However, an attempt was made to use these execution providers, but due to hardware compatibility issues, it was not possible to use them. Therefore, the CPU execution provider, which is the standard, was used. Compatibility issues will be studied in more detail in the cross-compiling Section.

The combination of ONNX for model conversion and ONNXRuntime for deployment provided a robust and flexible solution that met the needs of the project, enabling seamless and efficient integration of the Machine Learning models into the specific hardware environment. This approach ensured that the models could be adapted efficiently and effectively, taking advantage

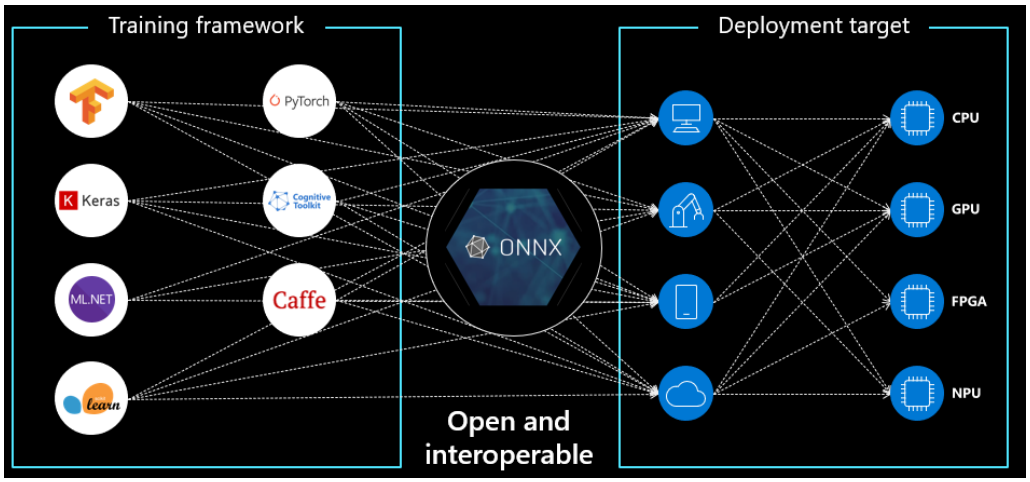


Figure 4.3: ONNX interoperability diagram with machine learning frameworks (From [28]).

of the ARMv7's specific capabilities, such as SIMD NEON instructions and vector floating point units (VFPv3 and VFPv4), while meeting the performance and compatibility requirements necessary.

4.2.3.3 Serialization Format of ONNX Models

The standard format for serializing models in ONNX is Protocol Buffers (protobufs), also developed by Google. Just like FlatBuffers used by TFLite 4.2.1, Protocol Buffers are designed to be a compact and efficient way to serialize structured data. They are widely used in various applications and services by Google and other organizations for network communication and data storage. Unlike FlatBuffers, data stored in Protocol Buffers generally needs to be deserialized (parsed) into a data structure in memory to be accessed. This can introduce a time and memory overhead, especially on devices with limited resources. Protocol Buffers support a wide range of programming languages and platforms, facilitating the integration of ONNX models in different development and execution environments. Similar to FlatBuffers, Protocol Buffers allow for the evolution of data schemas, making it easy to maintain and update models over time without breaking compatibility.

The main advantage of FlatBuffers over Protocol Buffers is access efficiency. FlatBuffers allow direct access to data without the need for prior parsing, which is a significant advantage in low-power or extremely resource-limited environments, such as microcontrollers. TensorFlow Lite is specifically optimized for mobile and embedded devices, utilizing FlatBuffers to minimize CPU and memory usage during model execution. ONNX, while supporting execution on various devices, may not be as optimized as TensorFlow Lite for low-power devices. ONNX has the advantage of being a widely supported open standard format, facilitating the exchange of models between different platforms and tools.

4.2.4 Pytorch Evaluation

The PyTorch framework was used mainly for converting PyTorch models to ONNX. This conversion capability is especially interesting because of the two different ways available to accomplish this task: TorchDynamo and TorchScript. Both approaches were explored during the project and proved to be viable, providing flexibility and efficiency in the conversion process.

TorchDynamo is a new conversion technique under development, designed to speed up the dynamic compilation of PyTorch models, enabling efficient conversion to the ONNX format. It stands out for its ability to optimize and speed up the conversion process, making it an attractive choice for developers looking to maximize the performance of their models in different execution environments. However, as this tool is still under development, some of its features may not be completely stable or available. TorchDynamo proved to be efficient at converting more complex models, compatible with a greater number of operations. However, the resulting models tended to be heavier and more complex, which can be a disadvantage depending on hardware restrictions.

To use TorchDynamo, you need to have ONNXScript installed. This is because ONNXScript provides the necessary infrastructure to interpret and convert PyTorch model operations into ONNX-compatible operations, facilitating efficient and accurate conversion.

On the other hand, TorchScript offers a more traditional approach to converting PyTorch models to ONNX. It allows models to be exported as static graphs, which facilitates integration and deployment on a variety of inference platforms. The use of TorchScript is widely supported and well documented, providing a robust and reliable alternative for model conversion.

During the development of the project, both techniques were evaluated and compared. TorchDynamo proved to be efficient in its compatibility with a wide range of operations, while TorchScript offered a more stable and widely tested approach over time. The comparison of these two tools made it possible to identify the advantages and disadvantages of each, contributing significantly to the flexibility and efficiency of the development and deployment workflow of PyTorch models for the ONNX format, while meeting the specific performance and compatibility requirements of the project.

4.2.5 Apache TVM Evaluation

Apache TVM was considered and studied as one of the frameworks for implementing machine learning models in this project. TVM is an open source machine learning compiler that transforms machine learning models into code ready for deployment on a variety of hardware platforms. Its main advantage is the ability to automatically optimize models for the target hardware, which can be extremely useful in AIoT applications where efficiency on a variety of devices is crucial.

Apache TVM is compatible with models from various frameworks such as PyTorch, TensorFlow Lite (TFLite) and ONNX. However, these models, which are usually in .onnx or .tflite formats, need to be converted into a .so file. This .so file not only contains the model, but also all the libraries needed to deploy the model on the specific hardware.

Despite the technical advantages offered by Apache TVM, the use of this framework encountered barriers due to Bosch's internal policies and processes. Bosch justified not using models in .so format due to issues related to code and internal processes, which make it difficult to integrate and use .so files. Furthermore, testing and validating models in a different cloud architecture presented significant challenges, making the use of the .so format unfeasible. Additionally, models in the .so format are not compatible with the tool used to visualize the models, which is also an important factor to consider. These difficulties combined led to the decision not to go ahead with using Apache TVM, despite its technical potential.

These justifications presented by Bosch were decisive in the decision not to go ahead with using Apache TVM, despite its technical potential.The need to adapt and validate the models in a format that better aligned with Bosch's existing practices and infrastructures led to the choice of other frameworks that do not require the creation of .so files for deployment.

4.2.6 Framework selection criteria

The selection of the frameworks used in this project was based on specific criteria, determined by the needs and limitations of the 32-bit ARMv7 hardware and the requirements of the Machine Learning operations needed. These criteria ensured that the frameworks chosen could offer an ideal balance between performance, compatibility and ease of implementation. Below are the main criteria considered when selecting the frameworks:

- **Hardware compatibility**
  The main criterion was compatibility with the 32-bit ARMv7 processor. It was essential that frameworks supported this type of hardware, taking full advantage of its capabilities, such as SIMD NEON instructions and vector floating point units (VFPv3 and VFPv4). Frameworks that could not run efficiently on ARMv7 were discarded.

- **Support for Necessary Operations**
  Another crucial criterion was support for the specific operations required by the project's Machine Learning models. Flow control and data transformation operations, such as While, If, ReduceSum, ReduceProd, and ReduceMax, needed to be supported by the frameworks. The lack of support for these essential operations resulted in the exclusion of some options, such as TensorFlow Lite Micro.

- **Ease of conversion and integration**
  The ease of converting models from other frameworks into the required format and integrating these models into the deployment environment was also a determining factor. Frameworks that offered efficient conversion tools, such as ONNX, were prioritized. The ability to integrate these conversions fluidly and without technical problems was vital to the efficiency of the workflow.

- **Stability and Maturity of the Framework**
  Framework stability and maturity were also considered. Frameworks that were already well

Table 4.1: Comparison of the Machine Learning Frameworks used

Framework	Used	Reason
TensorFlow Lite	Yes	Compatible with ARMv7 hardware, robust support for necessary operations, and ease of implementation.
TensorFlow Lite Micro	No	Lack of support for some essential operations such as While, If, ReduceSum, etc.
ONNX Runtime	Yes	Flexibility in converting models from other frameworks and efficiency in inferences on ARMv7.
PyTorch (TorchDynamo and TorchScript)	Yes	Mainly used to convert models to the ONNX format, exploring the feasibility of TorchDynamo and TorchScript.
Apache TVM	No	Incompatibility with the DFT operation, difficulty in integration with Bosch's internal processes, and challenges in validation on different architectures in the cloud.

established and widely tested, such as TensorFlow Lite and ONNX, were favored due to their proven reliability. Tools still in development, such as TorchDynamo, were evaluated but used with caution due to possible instabilities.

Based on these criteria, the frameworks that were considered viable and therefore used in the development of this project were TensorFlow Lite, chosen for its compatibility with ARMv7 hardware, robust support for required operations, and ease of implementation; ONNX and ONNXRuntime, selected for their flexibility in converting models from other frameworks and efficiency in performing inferences on ARMv7; and PyTorch, used mainly for converting models to the ONNX format, with both the TorchDynamo and TorchScript approaches being explored to assess their viability. These frameworks proved capable of meeting the specific requirements of the project, providing a solid basis for the development and deployment of efficient Machine Learning models on 32-bit ARMv7 hardware.

4.3 Cross-compiling

Cross-compilation is a crucial step in the process of developing and deploying Machine Learning models on devices with limited hardware, such as the 32-bit ARMv7. This process allows code to be compiled in one environment (usually a more powerful one, such as an x86 PC) and then executed in another (in this case, the ARMv7). Cross-compilation was essential to ensure that the models could be efficiently adapted and run on the specific hardware used in this project.

To carry out the cross-compilation, tools such as CMake and ARMv7-specific toolchains were used. Setting up the cross-compilation environment involved installing the appropriate toolchains and defining environment variables to direct the compilation to the target architecture.

In this section, the cross-compilation processes used for the different frameworks will be detailed. The discussion will start with TensorFlow Lite, explaining the specific steps and configurations needed to achieve a successful cross-compilation.

4.3.1 TensorFlow Lite cross-compilation

Cross-compiling TensorFlow Lite involved configuring a specialized CMake to handle the requirements of the ARMv7 architecture. A CMake file with the appropriate configuration was used to enable cross-compilation. Configuration involved defining the correct toolchain and preparing the necessary tools to ensure that compilation was targeted at the ARMv7 architecture. The toolchain, made up of the essential tools for cross-compilation, includes GCC (GNU Compiler Collection), G++ and sysroot.

GCC (GNU Compiler Collection) and G++ are essential compilers in this process. GCC is a compiler for the C programming language, while G++ is the version of GCC that compiles C++ code. Both are critical tools in cross-compilation, as they translate source code written in C or C++ into machine code executable by the ARMv7 processor. Specifically, version 8.3 of GCC and G++ was used due to device-specific configurations. Although there are more recent toolchains, version 8.3 is the latest to support the soft-float ABI (Application Binary Interface), which is required for this device. The latest versions only support bare-metal and hard-float, which are not compatible with the device's current configuration [15].

For this particular device, it is necessary to use the softfp option because of the way it is configured. The device is not configured to use hard-float, but it does have an FPU (Floating Point Unit) that can be used to perform calculations more efficiently. The softfp option allows the generated code to use the FPU for floating point operations, even if the soft-float ABI is used. This is done by passing the flags `-mfpu=neon-vfpv4` and `-mfloat-abi=softfp` during compilation.

The `-mfpu=neon-vfpv4` flag specifies that the compiler should generate code that uses the NEON FPU with support for the VFPv4 instruction set [10]. The `-mfloat-abi=softfp` flag indicates that the compiler should use the soft floating point ABI, where floating point variables are passed through integer registers, but still allows the use of the FPU to perform the actual floating point operations. This configuration is necessary to ensure that the code is compatible with the way the device is configured, making the most of the available FPU. Additionally, the `-funsafe-math-optimizations` flag was used and allows for aggressive mathematical optimizations that can improve performance, though they may result in slight differences in results due to reduced precision. The `-mfp16-format=ieee` flag specifies that the format of 16-bit floating point numbers should follow the IEEE standard, ensuring consistency and compatibility in handling these values.

The sysroot is a directory that contains a copy of the root file system of a specific system, including libraries and headers needed to compile programs for that system. During cross-compilation, the sysroot is used to provide a compilation environment that simulates the target system. This is crucial because it allows the compiler to find all the necessary dependencies (such as libraries specific to the target system) without needing these libraries to be present on the compilation system (usually an x86 PC). In our case, it was used a specific version of sysroot to ensure that all ARMv7 dependencies were available during the compilation process.

The command used for cross-compilation was:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
-DTFLITE_ENABLE_XNNPACK=OFF
-DCMAKE_TOOLCHAIN_FILE=`realpath ./toolchain.cmake`
```

The `DCMAKE_BUID_TYPE=Release` flag was used to ensure that the build was optimized for performance and size. The Release build type includes several optimizations that are not present in the Debug build type, as well as removing unnecessary debugging information, resulting in a smaller and more efficient executable. These optimizations are crucial for running TensorFlow Lite efficiently on the ARMv7 architecture, where computing resources are limited.

The `-DTFLITE_ENABLE_XNNPACK=OFF` flag was necessary because cross-compilation did not work correctly with XNNPACK enabled. XNNPACK is a highly optimized library for neural network inference that has specific compatibility with certain architectures and operating systems. In the case of ARMv7, XNNPACK is only compatible with Android, and as our device is not Android, there were compatibility issues. Disabling XNNPACK ensured that the compilation could be completed successfully, avoiding problems arising from these incompatibilities.

In addition, the libraries have been statically allocated. Static libraries are included directly in the executable during compilation, unlike shared libraries, which are linked at runtime. This results in a single, self-contained executable that includes all the necessary code. The advantage of static linking is that it can simplify deployment, as the executable doesn't depend on external library files being present on the device. However, this can also increase the size of the executable.

Now, with the executable created, you just need to send it to the device along with the model that the executable needs to work. With that, the cross-compilation process is complete, allowing TensorFlow Lite to run on the ARMv7 device as planned.

4.3.2 ONNX Cross-compilation

The cross-compilation of ONNX for the ARMv7 architecture presented some differences from the process used for TensorFlow Lite. Although the use of the toolchain (including G++ and sysroot) and the `-DCMAKE_BUILD_TYPE=Release` compilation flag were similar, there were some important specificities that needed to be addressed.

One of the main differences was the need to use shared libraries instead of static ones, due to easier cross-compilation. Shared libraries are linked at runtime, unlike static libraries, which are included directly in the executable during compilation. This results in an executable that relies on external library files present on the device, which can help reduce the size of the executable, but requires these libraries to be available in the runtime environment, this type of cross-compilation in principle does not noticeably alter the runtime of the models.

To build the executable, a Python script provided by ONNXRuntime's GitHub (`build.py`) was used. This script automates many of the configuration and compilation steps, but it was necessary to understand and pass the correct flags in order to cross-compile properly. The flags included

specifying the build type as Release, enabling the compilation of shared libraries (`-build_shared_lib`), the minimal build to reduce the size of the libraries (`-minimal_build`), omitting tests (`-skip_tests`), disabling exceptions (`-disable_exceptions`) and defining additional parameters specific to ARM compilation.

One important difference was that, during the initial cross-compilation attempt, the size of the resulting shared libraries was too large for the limited space available on the device. To solve this problem, the minimal build option (–minimal_build) was used, which significantly reduced the size of the libraries, making them more suitable for the device. Due to the use of minimal build, ONNX models that were previously .onnx files need to be converted to .ort format. This conversion applies various optimizations to the model, such as the elimination of unnecessary operations, merging of operations, quantization, precomputation of constants and optimized data organization. Although .ort files may be slightly larger than .onnx files, they are optimized for more efficient execution in ONNX Runtime, resulting in improved inference performance. To perform this conversion, ONNX Runtime provides a Python script [19]:

```
python -m onnxruntime.tools.convert_onnx_models_to_ort
```

Now, with the executable created, you need to send it to the device along with the template in .ort format and the shared libraries, specifically `libonnxruntime.so.1.18.0`. After transferring these files to the device, it is crucial to configure the environment so that the shared libraries are correctly located by the system when the executable is run. This can be done by setting the `LD_LIBRARY_PATH` environment variable to include the path where the `libonnxruntime.so.1.18.0` library is located. This setting ensures that when the executable is run, the system will find and load the shared libraries needed for ONNX Runtime to work correctly.

Two execution providers were utilized and tested during this process: the ARM Compute Library (ACL) and ARM NN. Both appeared to be very promising, offering hardware-specific optimizations that could potentially enhance the performance of ONNX models on ARM devices. However, after several attempts, it was discovered that these execution providers were exclusively designed for ARMv8 devices and not for ARMv7 [24]. As a result, due to these compatibility issues, it was not possible to use these execution providers effectively. Consequently, the CPU execution provider, which is the standard, was used.

With all these steps completed, the cross-compilation process is finished, allowing the ONNX Runtime to run on the ARMv7 device as planned.

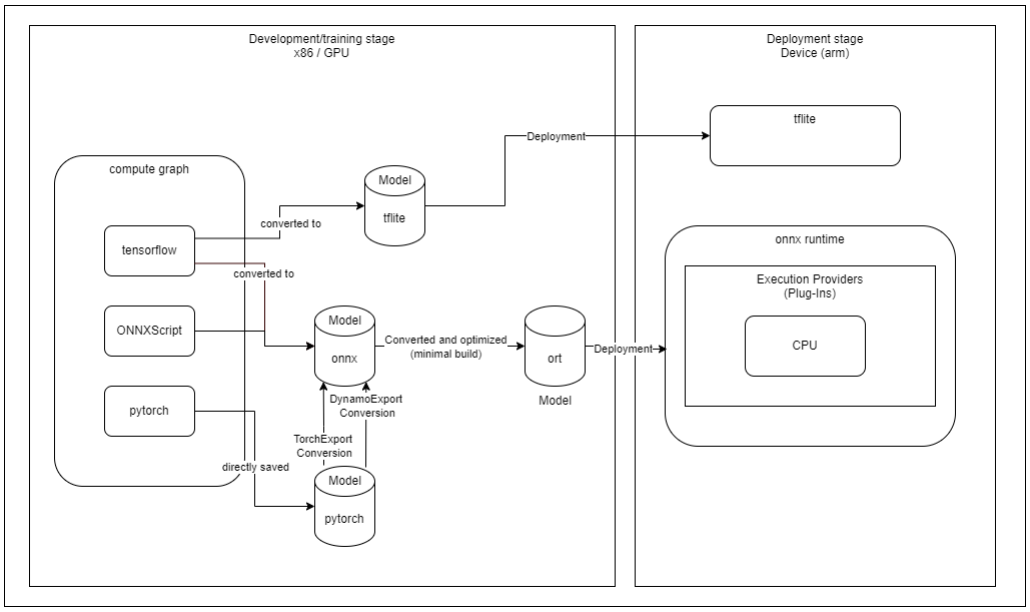


Figure 4.4: Final Frameworks Interoperability diagram for deployment.

After analysing the frameworks, the following diagram 4.4 represents what will be studied next.

4.4 Study of the Operations and Model

In the study of operations (ops) in the context of this project, it was essential to analyze and evaluate several specific operations and their impact on the compatibility and efficiency of Machine Learning models. In this study, the "flows" that represent the different paths a model can follow from its creation to deployment will be analyzed, involving different frameworks and conversion methods. These paths are crucial to ensuring that models can be efficiently adapted to the ARMv7 hardware environment, meeting specific performance and compatibility requirements. The operations chosen to represent the compatibility of the frameworks were ADD, SQRT, SORT, FFT, and LOOP, as they are crucial for the model being converted. Additionally, the model provided by Bosch was also studied. The Netron tool [18] was used to visualize the models generated for each operation, allowing for detailed analysis and comparison. Furthermore, the input tensors for all operations studied had a dimension of 8192 to replicate and study the compatibility of these large tensor dimensions in conversions and other frameworks.

4.4.1 Implementation Analysis of Adition Operation

The addition operation is one of the most basic and fundamental operations in any Machine Learning and signal processing model. It is used in a variety of contexts, from updating weights in learning algorithms to manipulating input data. In a normal model, numerous addition operations will be used, which makes it important to study them independently, without interference from

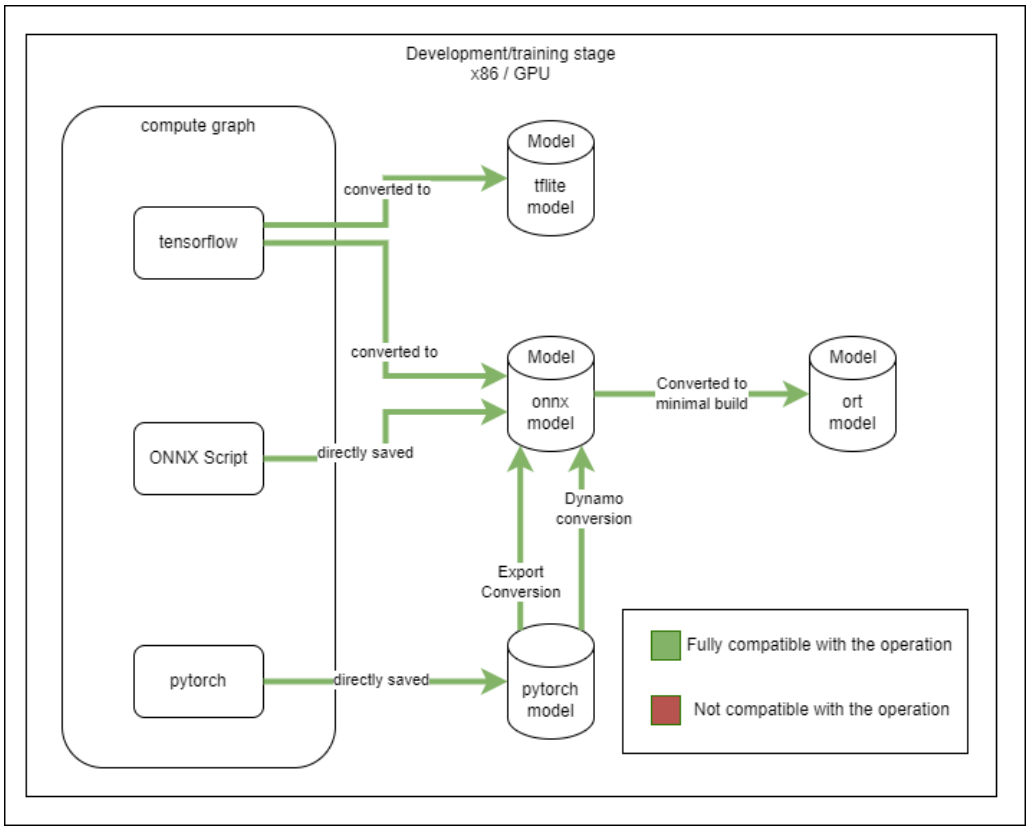


Figure 4.5: Framework compatibility diagram for operation ADD.

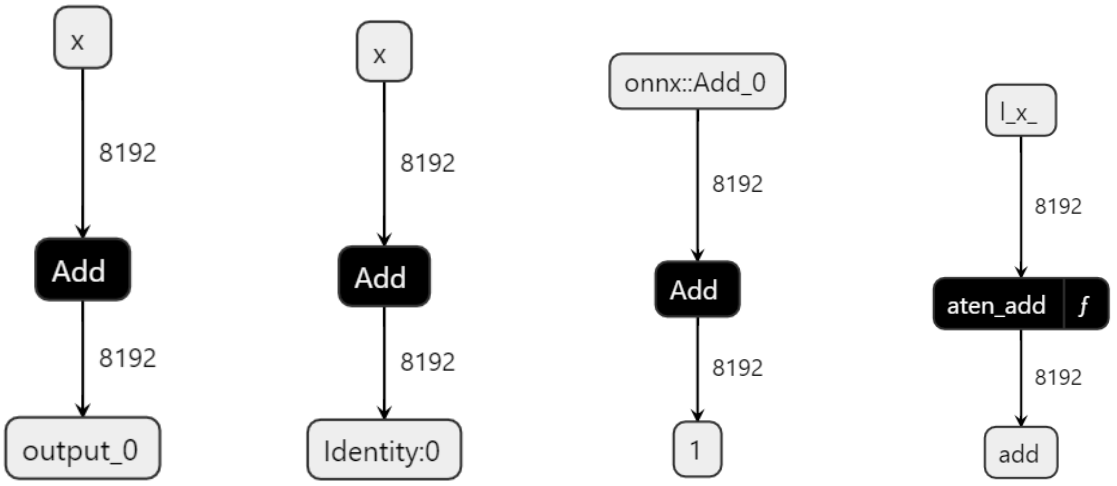
other nodes, in order to understand the time required to perform each of these operations. A small change in the efficiency of an addition operation can prove decisive in larger models, where many such operations are performed. This operation was used more for compatibility tests with the device, to see if it is possible to cross-compile simple models without problems, as well as to test the different performances of the runtimes when tested with simple models on the device. The model used for the ADD operation has been simplified as much as possible, where an input tensor is added to itself.

The ADD operation is compatible with all the converters used, as was to be expected given its simplicity and its fundamental nature in Machine Learning operations. The ADD operation is supported natively by all frameworks and converters, resulting in a straightforward and efficient conversion without the need for complex intermediate operations. This compatibility is demonstrated consistently in all conversions, ensuring that the expected behavior of the model is maintained in different execution environments.

Using Netron [18], the different models can be better illustrated:

The first three models (Figures 4.6a, 4.6b and 4.6c) are practically identical, as the ADD operation is simple, and there isn't much to change between the conversions. The only difference is in the names given to the inputs and outputs, which vary depending on the framework or conversion used.

On the other hand, model 4.6d, which shows the conversion from PyTorch to ONNX using



(a) TFLite conversion

(b) TensorFlow to ONNX conversion

(c) Pytorch to ONNX TorchScript conversion

(d) Pytorch to ONNX TorchDynamo conversion

Figure 4.6: Models for ADD operation

TorchDynamo, is more complex due to the export method used. The difference between the two conversions of PyTorch models to ONNX, as already explained in Section 4.2.4, is due to the different export methods: `torch.onnx.dynamo_export` and `torch.onnx.export`. The torch.onnx.export method, as seen above, is more traditional and widely used, resulting in a direct and optimized conversion that maps basic operations directly to corresponding ONNX operations. The dynamo_export method is a newer approach, designed to offer greater flexibility and support for dynamic and complex models. This method can keep some operations closer to PyTorch's original form (such as `aten_add` in this case), especially when it comes to operations that require runtime manipulation or that don't have a simple direct mapping to ONNX. This can result in ONNX graphs that appear more "raw" or less optimized, reflecting the original PyTorch model structure more faithfully, as evidenced by the presence of additional subgraphs and functions (f) in model of Figure 4.6d. When clicking on this f, additional subgraphs containing more tensors, calculations, and even more functions can be seen, as shown in Figure 4.7. While in one of the simplest operations this conversion method used these functions (f), in the next operations studied, more of these functions can be expected in the models translated from TorchDynamo, thus complicating and increasing the size of the model.

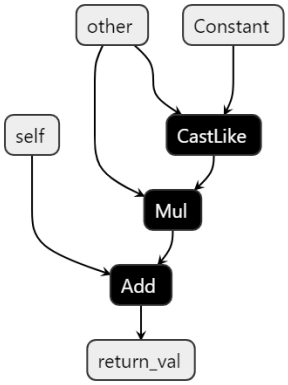


Figure 4.7: Function definition for ADD operation in ONNX model by PytorchDynamo conversion.

4.4.2 Implementation Analysis of Square Root Operation

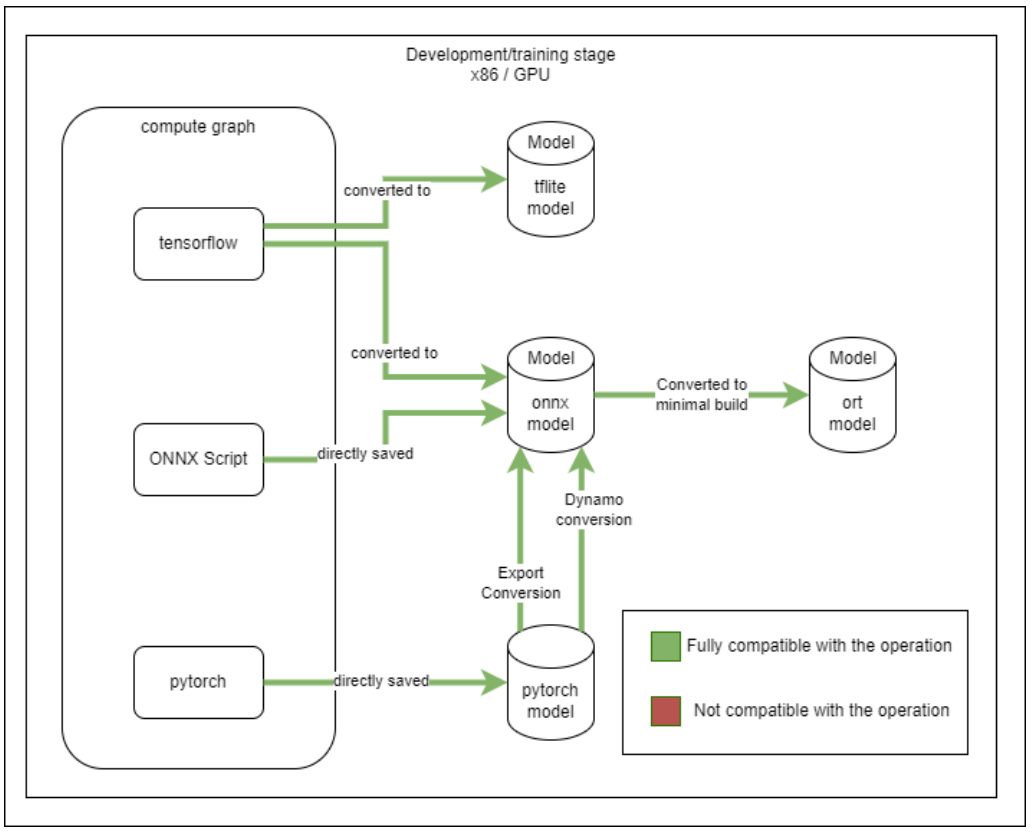
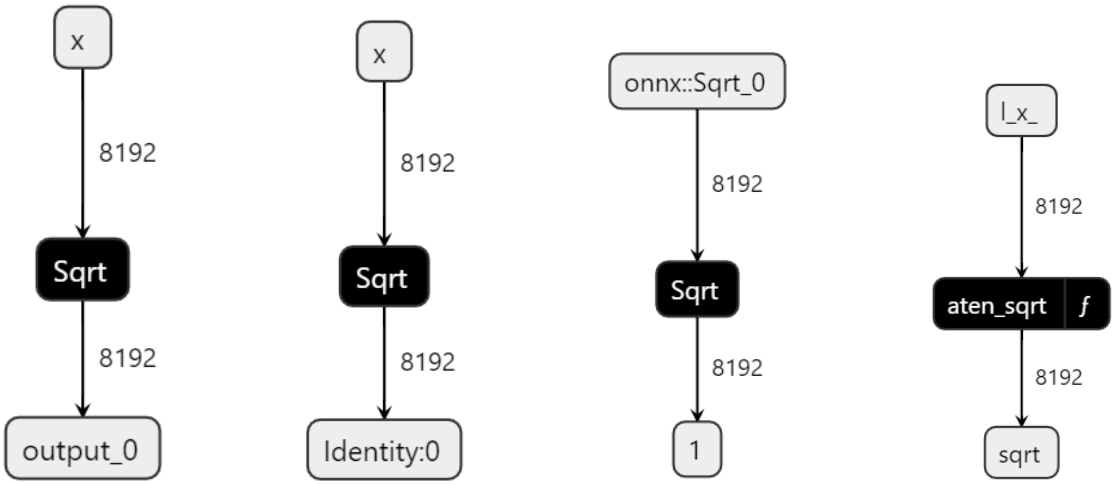


Figure 4.8: Framework compatibility diagram for operation SQRT.

The square root operation (SQRT) is essential in both Machine Learning models and signal processing. Used in a variety of contexts, it plays a key role in tasks such as data normalization, statistical calculations, and signal transformations. As in the previous subsection on the ADD operation, an assessment was made of a relatively simple and uncomplicated operation, which is



(a) TFLite conversion

(b) TensorFlow to ONNX conversion

(c) Pytorch to ONNX TorchScript conversion

(d) Pytorch to ONNX TorchDynamo conversion

Figure 4.9: Models for SQRT operation

nevertheless important, especially for assessing the compatibility and performance of the runtimes on the device. The model used for the SQRT operation was simplified as much as possible by performing the square root operation on an input tensor.

The results observed for the SQRT operation were consistent with those of the ADD operation. The simplicity of the operation allowed full compatibility between the different converters, with the only significant differences occurring in the names of the inputs and outputs, depending on the framework or conversion method used. Detailed analysis of individual operations, such as ADD and SQRT, is crucial to understanding the performance and efficiency of runtimes when running larger, more complex models where these basic operations are performed numerous times.

4.4.3 Implementation Analysis of SORT/TopK Operations

Sorting is a critical operation in many Machine Learning algorithms, especially those that involve analyzing sorted data or selecting the best candidates. The SORT operation uses the TopK operator in all of these frameworks, but not all of them manage to convert SORT into TopK. The SORT operation arranges all elements of a tensor in ascending or descending order and is important for tasks that require ranking or ordering data. The TopK operation returns the K largest elements of a tensor, along with their indices, and is crucial in many ML and data processing algorithms.

In the following diagram, each arrow indicates a sequence of steps and transformations that the SORT operation follows from its creation to deployment.

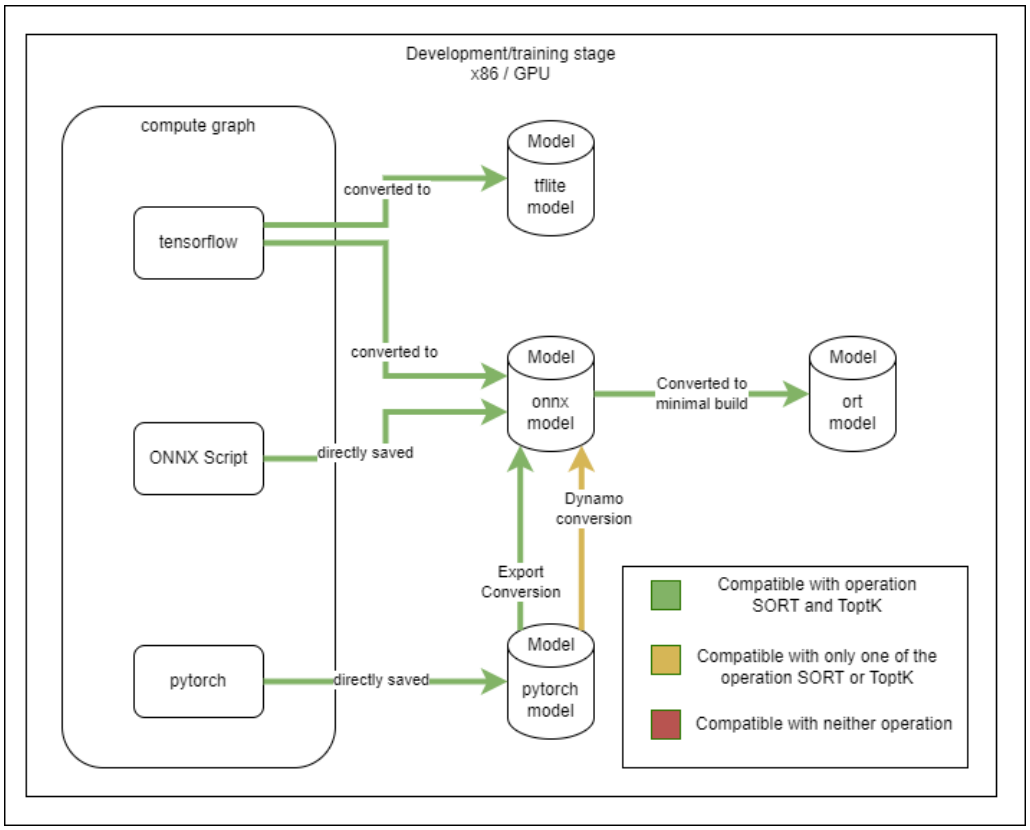


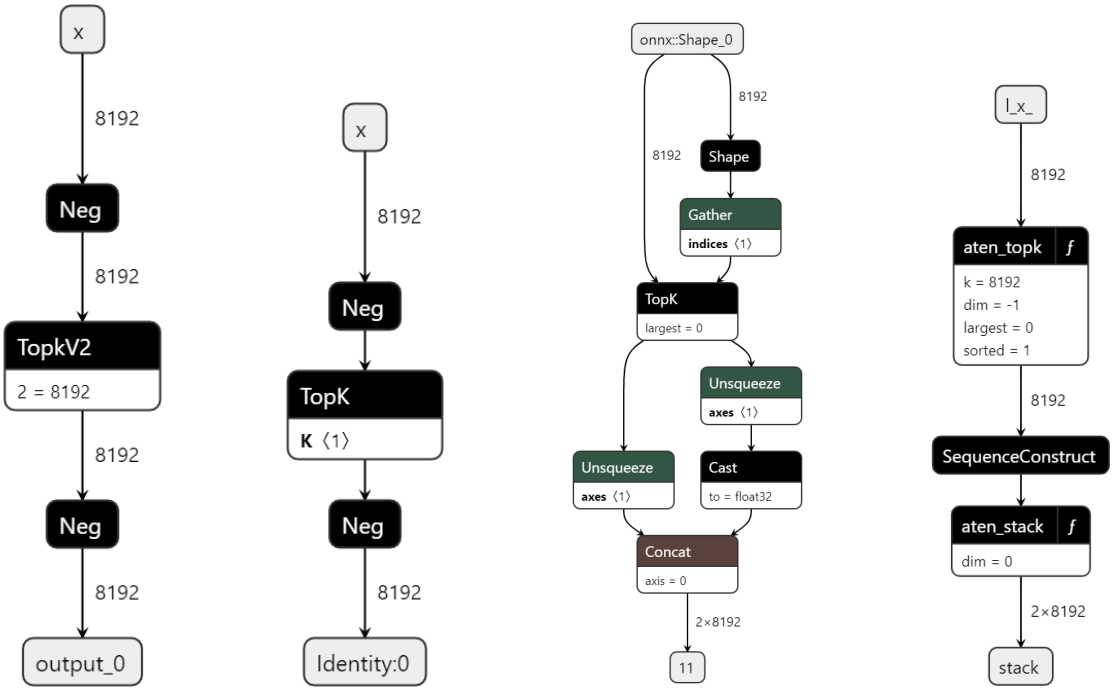
Figure 4.10: Framework compatibility diagram for operation SORT.

The SORT operation, which sorts the elements of a tensor, was first explored in TensorFlow. When the `tf.sort` operation is converted to TFLite, it becomes the TopKV2 operation. The TopKV2 operation is a more flexible version of TopK, which also exists in TFLite. This transformation is advantageous because TopKV2 retains the flexibility of TopK and is more efficient for certain applications.

When converted from TFLite to ONNX, the TopKV2 operation remains as TopK, allowing the model to take advantage of the efficiency of this operation during deployment.

However, when using PyTorch, the direct conversion of the SORT operation to ONNX via the TorchDynamo converter presents challenges. TorchDynamo cannot convert the SORT operation directly to TopK, requiring the TopK operation to be specifically used in the PyTorch model to ensure that the tensor ordering is correctly maintained during the conversion. For this reason, the conversion arrow appears in yellow in the diagram, indicating the problems encountered in this process. In the normal conversion using TorchScript, there were no problems with the transformation from SORT to TopK.

To better illustrate how the SORT operation is handled in different frameworks, the following figures show the detailed structure of the models after conversion, visualized using the Netron tool.



(a) TFLite conversion

(b) TensorFlow to ONNX conversion

(c) Pytorch to ONNX TorchScript conversion

(d) Pytorch to ONNX TorchDynamo conversion

Figure 4.11: Models for SORT operation

Looking at the models, the PyTorch conversions 4.11c 4.11d are noticeably larger, as can be seen from the visualizations of the models, especially the model generated by the Dynamo conversion 4.11d. The size of the models will be analyzed later in the data analysis and benchmarks Section, however, significant differences in complexity between the models can already be seen here. This is partly due to the fact that PyTorch, especially with Dynamo, as seen in the previous subsections on ADD and SQRT operations 4.4.1, 4.4.2, includes many intermediate operations and subgraphs to ensure the accuracy of the conversion and maintain the functionality of the model. However, in this case, there is an even greater difference in complexity due to the inherently more complex nature of the SORT operation compared to the previous operations. In contrast, TFLite is specifically designed to be lightweight and efficient on devices with limited resources, resulting in smaller, less complex models, so both the TFLite model 4.11a as well as the conversion of that model to ONNX 4.11b are simpler.

In Fig. 4.11a, referring to the TFLite conversion, a direct sequence of operations appears: Neg, followed by TopKV2, another Neg, and then the output. This simplicity reflects the lightweight and optimized nature of TFLite, which is designed to work efficiently on devices with limited resources. In Fig. 4.11b, referring to the conversion from TensorFlow to ONNX, the structure is similar to that of the TFLite conversion, with Neg, TopK, and another Neg operation, indicating that the complexity of the model remains comparable to that of TFLite.

In Fig. 4.11c, which shows the conversion from PyTorch to ONNX using TorchScript, the

model is significantly more complex. The diagram includes additional operations such as Shape, Gather, Unsqueeze, Cast, Concat, in addition to the main TopK. This additional complexity is introduced to ensure that all intermediate PyTorch operations are correctly represented and maintained during conversion while preserving the full functionality of the original model. In Fig. 4.11d, which shows the conversion from PyTorch to ONNX using TorchDynamo, the resulting model is the most complex of all. The diagram shows several additional operations and the use of functions f, as indicated by the blocks next to `aten_topk`. When clicking on the f, additional subgraphs containing more tensors, calculations, and even more functions can be seen, as shown in the second Fig. 4.16. This contributes significantly to increasing the size and complexity of the model. The presence of these functions f indicates that there are additional calculations and operations encapsulated that are necessary to maintain the accuracy and functionality of the model after conversion.

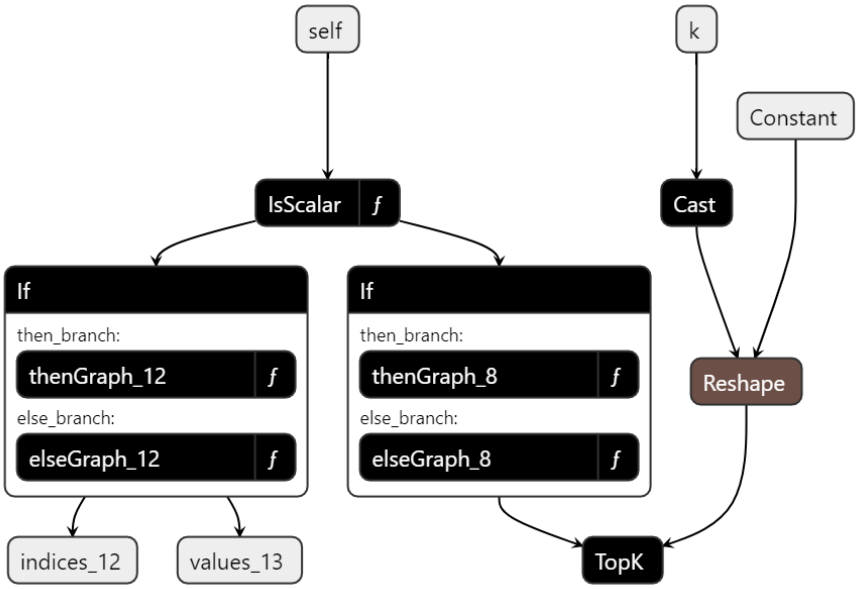


Figure 4.12: Function definition of SORT model PyTorchDynamo.

4.4.4 Implementation Analysis of Discrete Forrier Transform Operation

The Discrete Fourier Transform (DFT) operation and its more efficient version, the Fast Fourier Transform (FFT), are crucial in both Machine Learning models and signal processing. Used in a variety of contexts, they play a key role in tasks such as frequency analysis, signal filtering, and image processing. As with the ADD and SQRT operations, an evaluation of these operations was carried out to check the compatibility and performance of the runtimes on the device.

The Discrete Fourier Transform (DFT) is computationally intensive with a time complexity of $O(N^2)$. This makes it less suitable for applications requiring fast processing of large datasets. The Fast Fourier Transform (FFT) improves upon the DFT by reducing the time complexity to $O(N \log N)$. This significant reduction is achieved through algorithmic optimizations that exploit

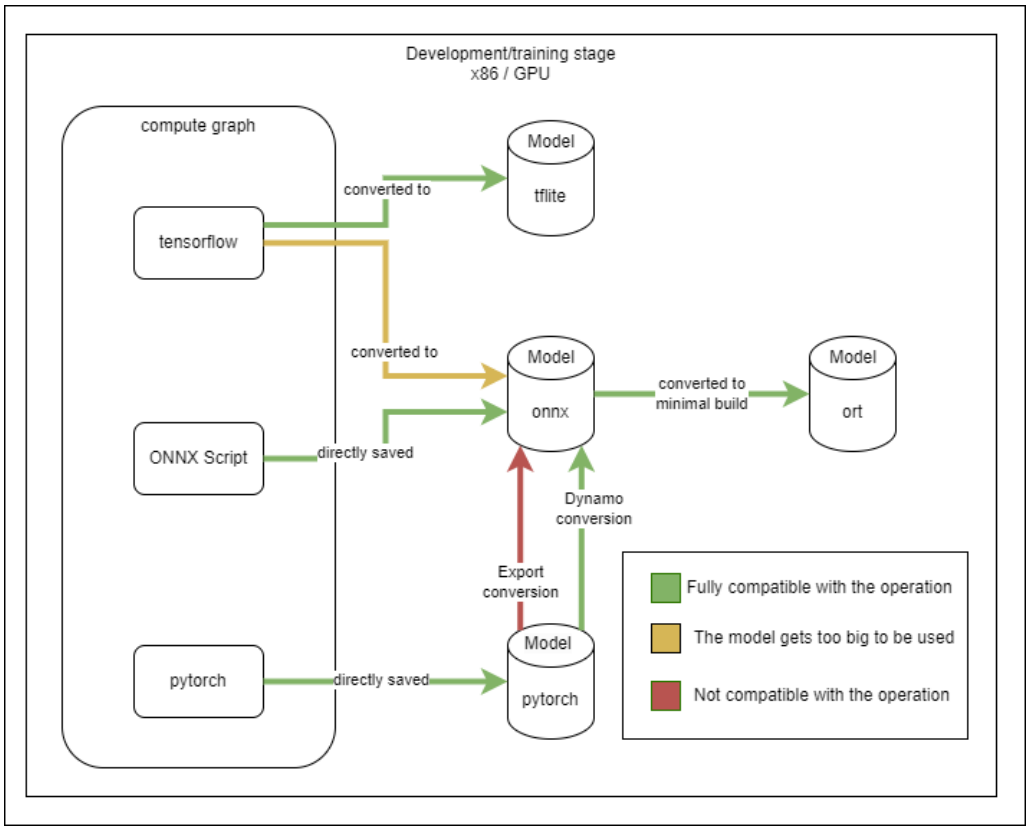
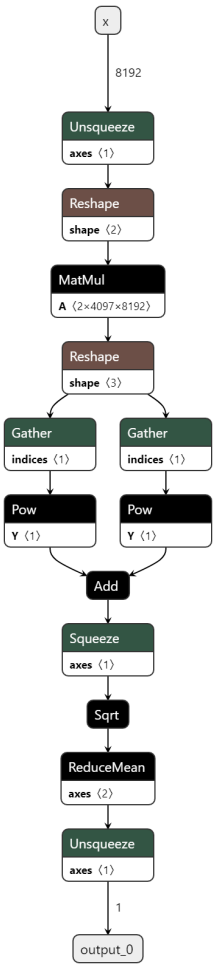


Figure 4.13: Framework compatibility diagram for operation FFT/DFT.

symmetries and redundancies in the DFT calculations. As a result, the FFT is much faster than the DFT, making it preferable for real-time signal processing and large-scale data analysis.

The FFT/DFT operation generated many more compatibility problems, especially on the ONNX Framework side. When performing the transformation from TFLite to ONNX and from PyTorch to ONNX using the normal PyTorch converter, it was discovered that the FFT node in ONNX did not exist, unlike in TFLite. This is due to the fact that ONNX is more focused on neural networks, and so the FFT operation does not yet exist. When this conversion was done, the resulting models were extremely large, reaching around 260000 KB. This made them useless for this project, as it was not possible to use them on the hardware in question due to the limited memory available.



(a) TFLite conversion

(b) Matmul tensor input

Figure 4.14: FFT ONNX conversion representation.

This huge expansion of the model occurs because, during the conversion, the converters did not directly transform the FFT node from the TFLite to the DFT node (which exists in the most recent versions of ONNX). Instead, the operation has been broken down into several operations, the most relevant of which is the matrix multiplication operation (MATMUL) with initializers that are pre-calculated matrices to simulate the FFT operation 4.14a. The resulting model includes an input tensor of size $2 \times 4097 \times 8192$, filled with values between 0 and 1 4.14b. These initializers are large matrices that contain the coefficients needed to perform the FFT operation. The presence of these large tensors with many values is what contributes significantly to the inflated size of the model.

The use of MATMUL operations instead of a single DFT operation is due to two main factors. First, computational efficiency plays a crucial role. In many cases, the MATMUL operation can be more efficient or better supported by hardware and optimization libraries such as BLAS and cuBLAS than a dedicated DFT operation, especially when the input tensors are smaller. Matrix multiplication operations (MATMUL) are highly optimized in many machine learning libraries

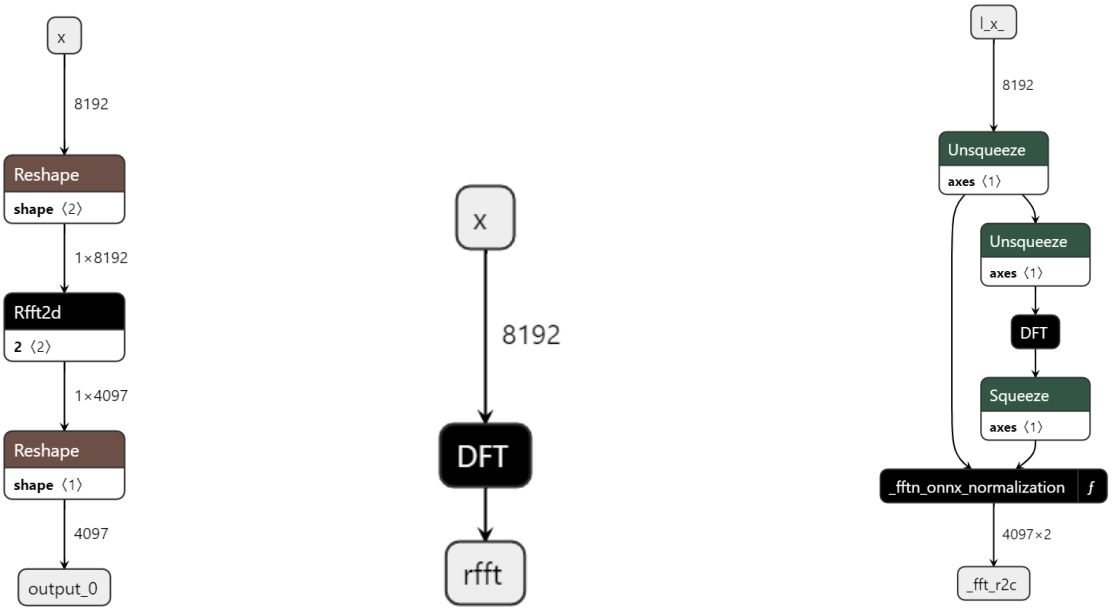
and can be accelerated by GPUs and other hardware architectures, making them an efficient choice for implementing FFT transformations.

In addition, the use of more primitive operations, such as MATMUL, increases the compatibility and interoperability of the ONNX model. Not all implementations of ONNX Runtime or other inference runtimes can support all the latest and most advanced operators, while basic operations such as MATMUL are widely supported. Converting directly to DFT can introduce subtle differences in the implementation of the transformation, depending on how the DFT operation is implemented in the specific backend. Using MATMUL and other more primitive operations can allow for a more faithful representation of the original FFT transformation.

The next step was to find a solution to this problem, as the DFT operation was quite important for the required model. The first phase was to discover that there is a way to write ONNX models directly and write node by node using ONNXScript. Subsequently, a method called "graph surgeon" can be used to add the required node to the model in question. This method involves modifying already created models to add or remove nodes and tensors. However, this method proved to be quite laborious and not very user-friendly.

Another solution found was the use of DynamoExport. It was during this exploration that the DynamoExport converter was discovered. Since DynamoExport inherently works with ONNXScript, it uses the required DFT node during the translation, resulting in a model that is not as large as those produced by other conversion methods. These are the two methods found to work around the problem of this operation.

The resulting models were then just the TFLite model, the model generated by DynamoExport conversion, and the model created using ONNXScript. These models will now be analyzed using Netron to better assess their structure.



(a) TFLite conversion

(b) ONNXScript model creation

(c) Pytorch to ONNX TorchScript conversion

Figure 4.15: Models for FFT/DFT operation

In the first Fig. 4.15a, TFLite uses the Rfft2d node. This node is specifically designed to perform the FFT transformation on two-dimensional inputs. The Rfft2d operation requires the input tensor to have two dimensions, which explains the need to use a tensor that is a perfect square. The FFT is transformed at this particular node because the Rfft2d operation is optimized to work with two-dimensional inputs, allowing efficient application of the Fourier transform in both directions of the tensor. This ensures that the operation is carried out quickly and accurately, but it also limits flexibility in model design, since the Rfft2d operation doesn't work properly if the input doesn't have the specific shape required. This is one of the reasons why you can't use an input of 8000 points and have to use the nearest square, in this case 8192.

In the Fig. 4.15b, the model created with ONNXScript shows a simple structure, reflecting the still-developing nature of this language. However, the presence of two Unsqueeze nodes is noticeable. These nodes are needed to adjust the dimensions of the input tensor before applying the DFT operation, ensuring that the shape of the tensor is compatible with the subsequent operation.

Finally, the DynamoExport conversion Fig. 4.15c shows a more complex structure, as seen in other cases. The `_fftn_onnx_normalization` subfunction is responsible for normalizing the DFT output values, ensuring that the resulting data is scaled appropriately. This normalization process helps maintain the consistency and interpretability of the transformed data.

4.4.5 Implementation Analysis of Loop Model

The LOOP operation was studied due to its importance in the model being studied for compatibility with different frameworks. In Machine Learning and signal processing, loops with if conditions are crucial for various tasks, such as iterations and condition checks over input data. Loops allow models to handle sequences and repetitive computations efficiently, which is fundamental for tasks like RNNs, LSTMs, and other iterative algorithms.

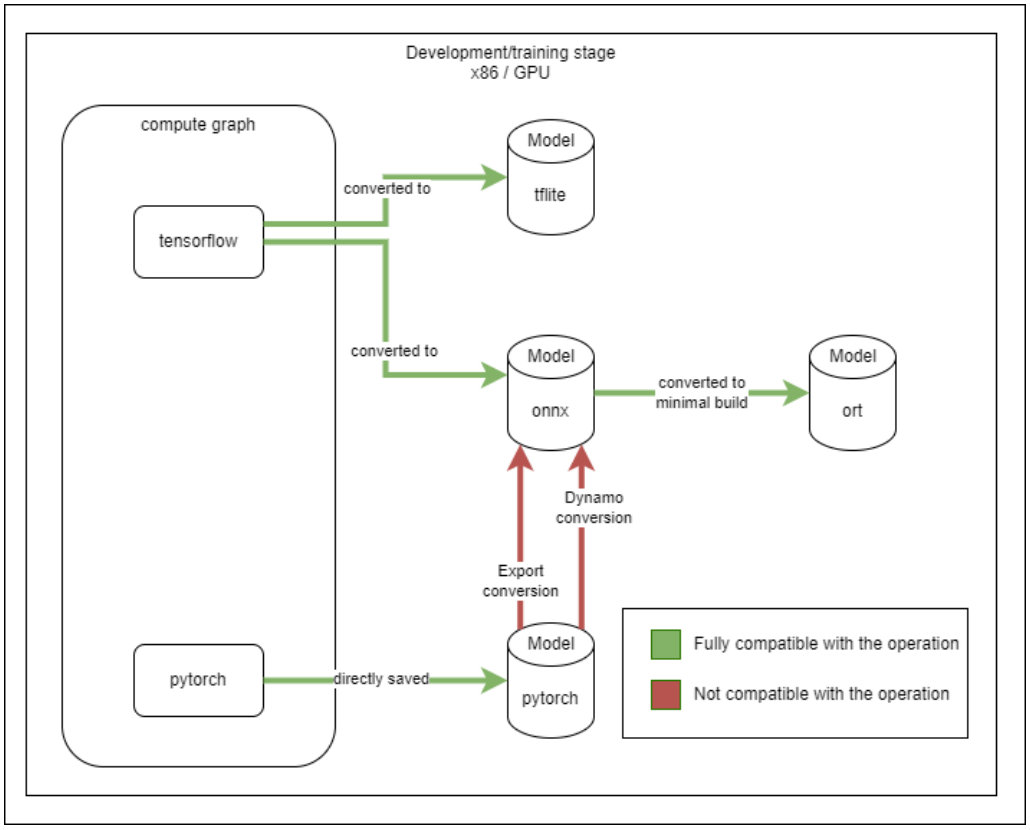


Figure 4.16: Framework compatibility diagram for LOOP Model.

The first loop developed was a simple loop that traversed the input tensor of 8192 points between 0 and 1 and compared whether the values were greater than 0.5. The resulting TensorFlow models were successful, producing the TFLite model and the TensorFlow to ONNX conversion. However, the resulting PyTorch models presented problems. When the models were generated, the result was an “empty” model, which only returned the number of values below 0.5 of the example tensor entered, i.e., it was a model that only returned a constant, as shown in the Fig. 4.17. This issue arises because PyTorch’s tracing mechanism struggles with data-dependent control flows, leading to the loss of dynamic operations during export.

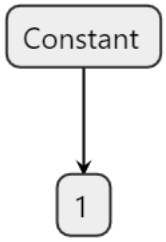


Figure 4.17: LOOP model PyTorchScript.

Due to this problem, another loop model with a condition was created, but this time forcing a 'graph break' when comparing values within the tensor, specifically the current value in the loop with the previous one. If the current value is greater than the previous one, it is divided by 2. This way, the model couldn't simply return a constant. However, the resulting PyTorch models could not be exported to ONNX, resulting in an export error and no model being generated. This is due to a fundamental limitation of PyTorch's torch.export method, as explained in the official documentation:

> "Although torch.export shares components with torch.compile, the main limitation of torch.export, especially when compared to torch.compile, is that it doesn't support 'graph breaks'. This is because dealing with graph breaks involves interpreting the unsupported operation with Python's default evaluation, which is incompatible with using export. Therefore, in order for the template code to be compatible with torch.export, it is necessary to modify the code to remove 'graph breaks'." [23]

Graph breaks are problematic because they introduce interruptions in the linearity of the computational graph. During model execution, the data flow does not follow a continuous and predictable path but is affected by data-dependent conditions and loops. In frameworks such as PyTorch, these interruptions are managed dynamically, allowing flexibility during the execution of training and inference. However, when it comes to exporting the model to a static format such as ONNX, these interruptions become problematic. ONNX requires a static, well-defined computational graph to ensure compatibility and portability between different frameworks and execution environments. Therefore, graph breaks introduced by conditional structures and data-dependent loops make direct export to ONNX unfeasible without significant modifications to the model code.

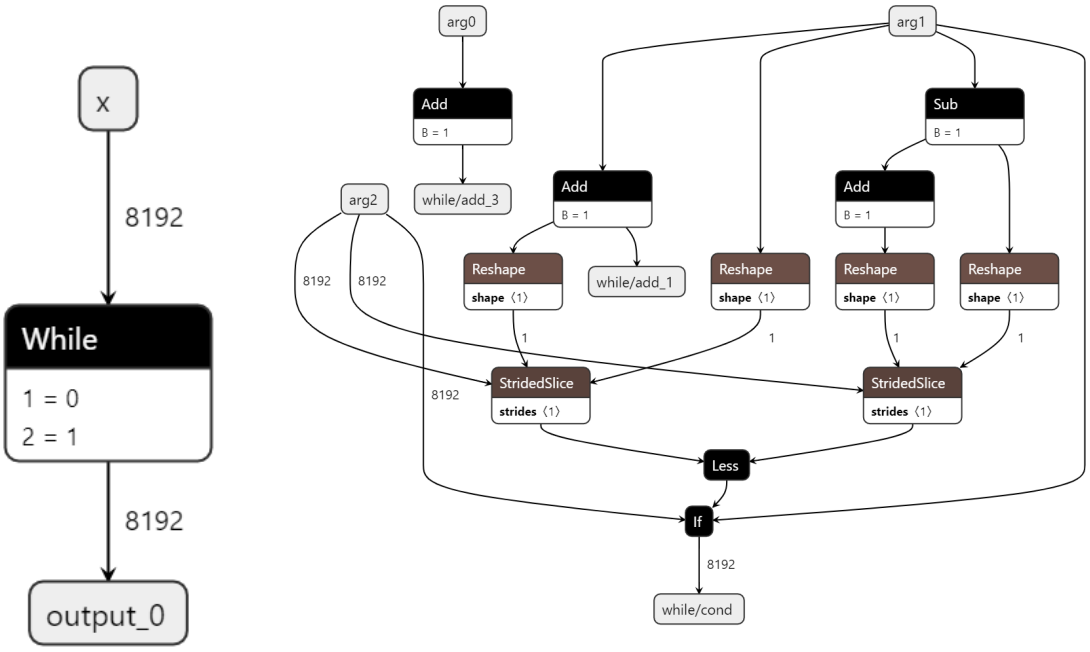
A graph break is necessary in cases such as data-dependent flow control, which makes conversion to ONNX unfeasible without significant modifications to the model's code to avoid such breaks. For this particular work, flow controls are essential, as they are critical to the model's logic. These flow controls include graph breaks, which are interruptions in the linearity of the computation graph, introduced by conditional structures and loops that depend on the data.

In this case, the loops with if conditions in the model also introduce graph breaks, making it impossible to convert directly to ONNX using torch.export. The presence of these data-dependent flow controls is fundamental to the model's functionality but, at the same time, represents a significant obstacle to direct export to ONNX without rewriting the code. Therefore, converting

conditional loops to ONNX is not feasible without removing or restructuring these graph breaks in the model's code, something that is neither practical nor desirable.

Due to the problems encountered, two models remained for analysis, the TFLite and ONNX converted from Tensorflow models. It was decided to focus on the second model, where the current value is compared with the previous value to determine whether the operation is carried out or not. This model presents a more complex flow control and can provide more interesting data for analysis.

To better understand the structure of these models, visualizations were generated using the Netron tool. The following Figures illustrate how the cycle used works, as well as the different branches within the cycle depending on whether the condition has been met or not. Let's start by analyzing the TFLite model:

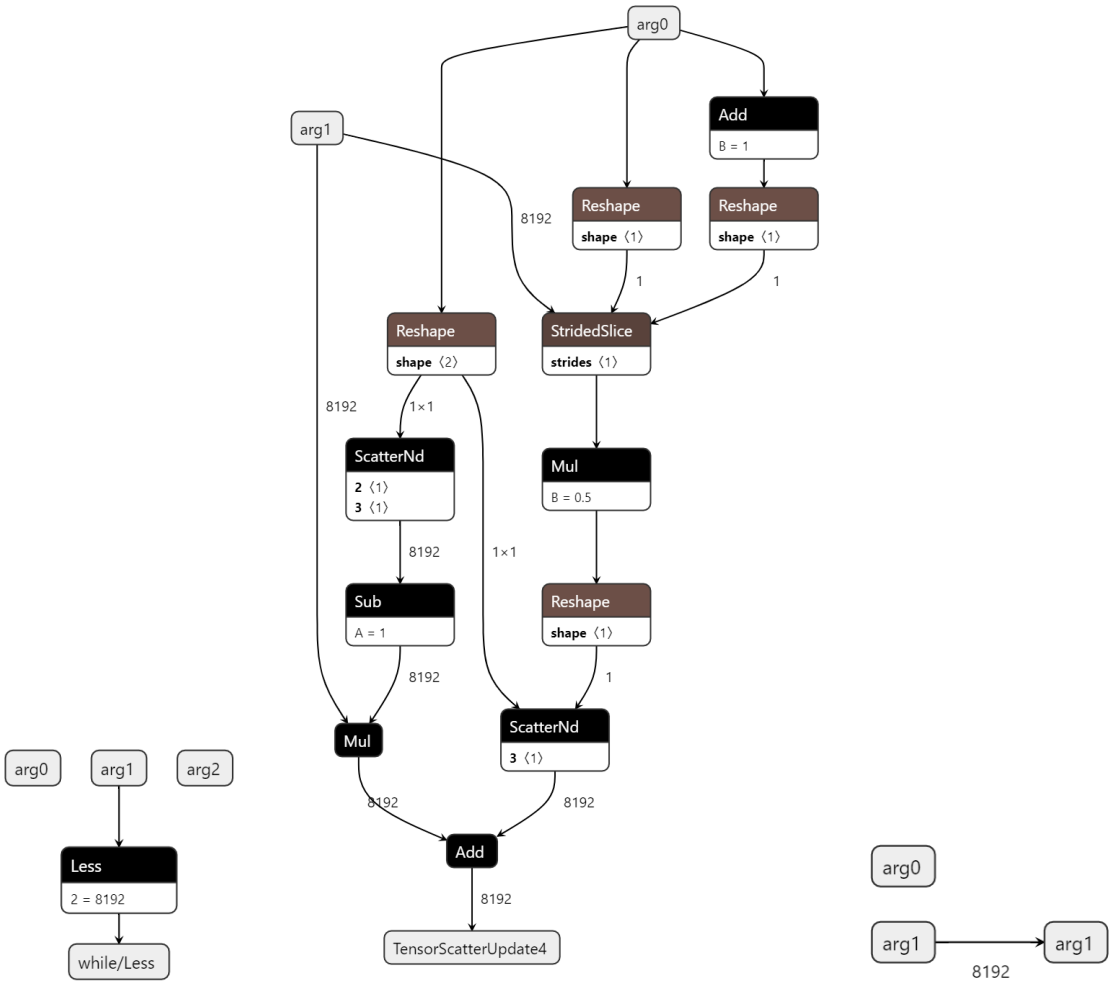


(a) Loop main node

(b) Loop body

Figure 4.18: Model for LOOP operation in TFLite (Node and Body)

In the first Fig. 4.18a, the principal node of this model, the 'while' node, can be seen.. Within this cycle, the logic is defined by comparing the current value with the previous value of the tensor. If the current value is greater than the previous one, it is divided by 2. Otherwise, the value is just passed on to the output tensor. Intermediate operations involve format adjustments and indexing to ensure that the comparison and updates are applied correctly, as can be seen in the Fig. 4.18b. This Figure represents what is inside the 'while' node, often referred to as the "Loop body."



(a) Loop end condition (b) True condition (c) False condition

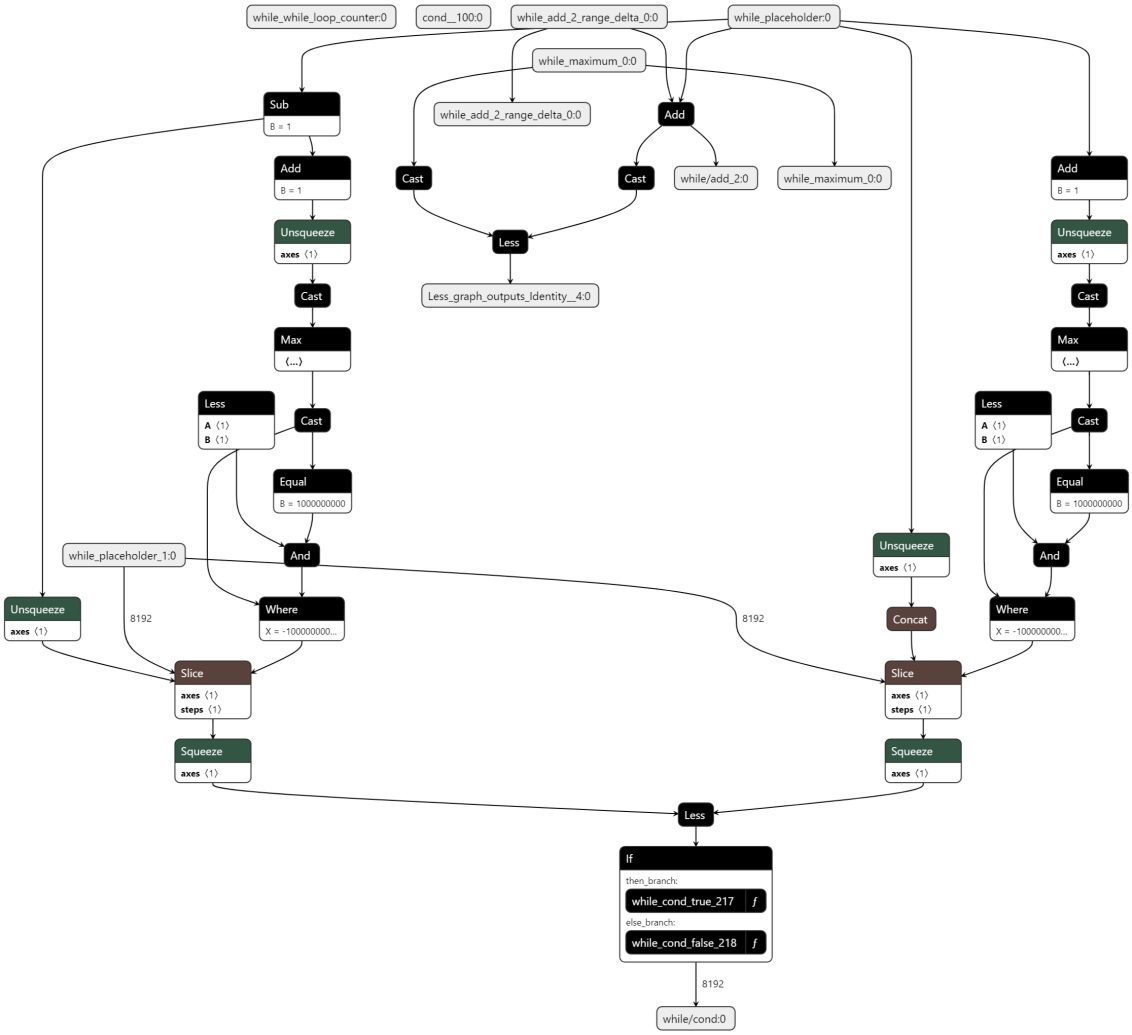
Figure 4.19: Model for LOOP operation in TFLite (Conditions)

The Fig. 4.19a shows the condition of the loop. The condition determines whether the loop continues or ends. The output of this condition is a boolean that controls the execution of the loop.

The Fig. 4.19b represents the scenario where the condition is true, meaning the current value is greater than the previous value. In this case, the current value is divided by 2 and updated in the output tensor. This ensures that values meeting the condition are appropriately processed before being passed forward. The intermediate nodes involved in this process handle reshaping and multiplication operations necessary to update the tensor correctly.

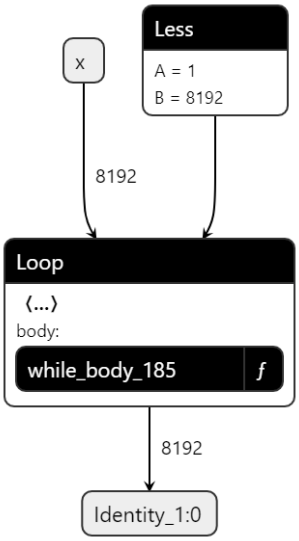
In contrast, the Fig. 4.19c illustrates the false condition, where the current value is not greater than the previous value. Here, the value is passed directly to the output tensor without any modifications. This path is simpler, involving fewer operations, as the condition was not met and thus, no additional processing is required.

Analyzing now the ONNX model that was converted from TensorFlow:

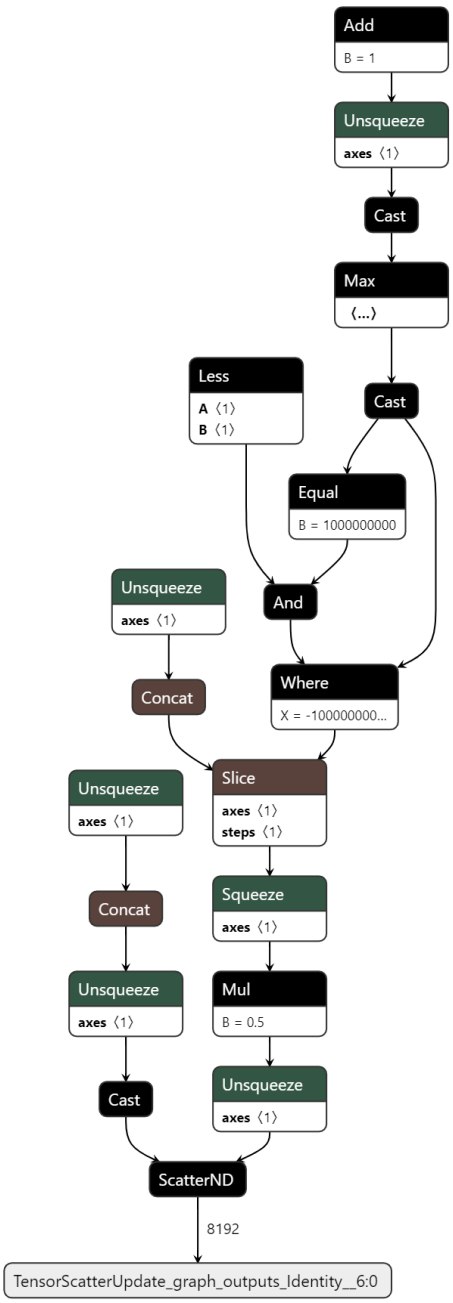


(a) Loop body

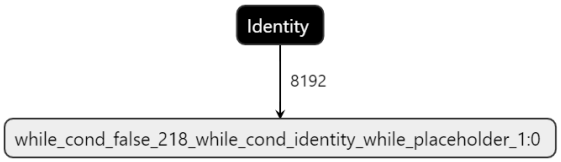
Figure 4.20: Model for LOOP operation in ONNX conversion



(a) Loop node and end condition



(b) True condition



(c) False condition

Figure 4.21: Model for LOOP operation in ONNX conversion - Part 2

The Fig. 4.21a represents the main node of the loop, and the image also shows the loop termination condition in the ONNX model. This condition is represented by the “Less” node in the main graph, which checks whether the current index is less than 8192, thus determining the continuity or termination of the loop. The result of this check is a boolean that controls the execution of the loop. In comparison, in the TFLite model, this termination condition is checked in a subgraph within the loop node, while in the ONNX model, this check occurs directly in the main graph, simplifying the loop structure.

The Fig. 4.21b illustrates the path followed within the loop when the specific condition of the current value of the tensor being greater than the previous value is met. In this case, the current value is divided by 2, and the output tensor is updated with this new value. The intermediate nodes include operations such as “Add”, “Cast”, “Unsqueeze”, “Mul”, and “ScatterND”, which are used to manipulate and update the tensor as required. These operations are necessary to ensure the tensor’s dimensions and data types are compatible with subsequent computations, thus maintaining the integrity of the loop’s logic. This condition is distinct from the loop termination condition and is evaluated during each iteration of the loop to determine the operations to be performed on the tensor.

On the other hand, the Fig. 4.21c represents the path followed when the loop condition is not met, i.e. when the current value of the tensor is not greater than the previous value. In this scenario, the current value is simply passed to the output tensor without modification. This ensures that the loop continues to process the next value in the input tensor. These two conditions are represented as subgraphs in the while_body within the "if" node represented in the Fig. 4.20.

Comparing the two models, you can see that the ONNX model is more complex than the TFLite model. Both models result from conversions, but the TFLite model tends to be more simplified due to the fact that it is converted directly within the same framework, i.e. from TensorFlow to TFLite. This internal conversion allows the TFLite converter to be more efficient and optimized, resulting in a more straightforward and less intricate structure. On the other hand, the conversion to ONNX, especially when originating from frameworks such as TensorFlow, can be more complex due to the need to maintain compatibility between different frameworks and standards. This means that ONNX preserves more detail and intermediate operations, resulting in a more detailed and often more complex representation of the model.

4.5 Study of the Collision Detection Model

In this section, the details of the machine learning model provided by Bosch will be explored, which is a centrepiece of this study. The model’s main function is to analyse accelerometer data to determine whether a collision has occurred. This capability is crucial for applications in the automotive industry, especially to improve safety by accurately identifying collisions.

4.5.1 Model Overview

The model supplied by Bosch processes data from accelerometers, which are integral components of Inertial Measurement Units (IMUs). These sensors capture detailed information about a vehicle's movement and orientation. By analysing this data, the model can assess the likelihood of a collision, distinguishing between real collisions and other events such as sudden stops.

The use of accelerometer data is particularly advantageous because it allows for real-time analysis and decision-making, which is essential for implementing effective safety measures in vehicles. The model's ability to accurately detect collisions can significantly improve the safety features of modern cars, contributing to the overall goal of increasing passenger protection and vehicle integrity.

4.5.2 Purpose, Practical Relevance and Conversion Challenges

The main objective of using this model is to explore its performance and compatibility in different machine learning frameworks, namely TensorFlow Lite, ONNX and PyTorch. This analysis between frameworks aims to identify the most efficient and reliable framework for implementing the model in a real environment with limited resources.

However, the model received was originally developed in TensorFlow, so there were two main options to explore in this project: converting to TensorFlow Lite (TFLite) or to ONNX. However, the conversion to ONNX faced significant problems that made this translation impossible. Although promising, using the ONNX translation method was not successful, so only the conversion to TFLite was carried out. This conversion was easy to carry out, being similar to what had been done for other previous models.

To justify the impossibility of converting the model from TensorFlow to ONNX due to the specific error encountered, it is important to understand and explain the differences between the implementations of complex mathematical operations, particularly the Fast Fourier Transform (FFT), in the two frameworks.

The primary issue lies in the differences in how TensorFlow and ONNX handle operations involving complex numbers. TensorFlow supports complex operations such as RFFT (Real FFT) and FFT, which can directly manipulate the real and imaginary parts of complex numbers. These operations are essential for models that work with frequency domain data, enabling detailed and efficient analysis of signal components.

In contrast, ONNX is a model interoperability platform that does not inherently support all TensorFlow-specific operations. Notably, operations that handle the separate real and imaginary parts of complex tensors ('Real' and 'Imag') are not equivalently supported in ONNX. This discrepancy became evident when an error occurred during the conversion process: `"ValueError: make_sure failure: Current implementation of RFFT or FFT only allows ComplexAbs as consumer not 'Real', 'Imag'"`. This error indicates that the ONNX conversion expects the output of RFFT or FFT operations to be consumed directly as complex absolute values (ComplexAbs), rather than as separate real and imaginary components.

The context of this error is crucial. ComplexAbs refers to the calculation of the magnitude of a complex number, an operation that ONNX supports. However, the model from TensorFlow explicitly requires access to the individual real and imaginary parts of the complex output, which ONNX does not directly facilitate. This leads to a fundamental incompatibility because ONNX lacks a mechanism to represent or process these components separately in the same way that TensorFlow does.

The conversion process failed because model conversion necessitates that all operations are translatable from the source framework to the target framework. TensorFlow's native ability to handle the separate real and imaginary parts of complex numbers through its specific operations does not have a direct equivalent in ONNX. This specific requirement of the model, to use real and imaginary parts separately, is supported by TensorFlow's operations but is not replicated in ONNX, which uses a more generalized approach for some advanced mathematical operations.

4.5.3 TFLite Model Structure

The conversion of the Bosch collision detection model to TensorFlow Lite format was a crucial step to ensure that the model could run efficiently on resource-constrained devices. This process began with an initial analysis of the model to understand its architecture, including the types of layers used, the complexity of the operations, and the overall size of the model. This preliminary analysis was essential to identify potential issues that could arise during the conversion process and to plan the necessary optimizations.

The conversion itself was performed using the TensorFlow Lite Converter, a tool provided by TensorFlow specifically designed to optimize models for mobile and embedded devices. This tool translates the high-level TensorFlow operations into the more efficient TensorFlow Lite operations. The conversion process was initiated with the `tflite_convert` command, which involves specifying the input and output arrays of the model and generating a TensorFlow Lite FlatBuffer file. This file format is highly optimized for mobile and embedded applications due to its compact size and efficient memory usage.

The structure of the converted TensorFlow Lite model consisted of several key layers, including convolutional layers, dense layers, and activation functions. Each layer was carefully analyzed and adjusted to ensure compatibility with TensorFlow Lite. Unsupported operations were replaced with equivalent supported operations, and the overall architecture was maintained to ensure the model's functionality remained intact.

After conversion, the TensorFlow Lite model underwent thorough testing on the ARM Cortex-A7 hardware. This testing phase involved measuring execution time, inference time (for the model), memory usage, and overall system performance to ensure the model met the required standards. Additionally, real-world scenarios provided by Bosch were used to validate the model, ensuring it could reliably detect collisions and perform effectively in practical applications.

In summary, the conversion of the Bosch collision detection model to TensorFlow Lite format was a detailed and methodical process that involved addressing compatibility issues and validating the model on target hardware. This conversion enabled the deployment of an efficient and effective

machine learning model capable of running on resource-constrained devices, providing valuable insights and enhancing safety in mobility services.

Chapter 5

Benchmarking Results of ML Frameworks in Resource-Constrained Environments

In this section, the results of the evaluation of the selected machine learning frameworks are presented and discussed. The metrics analyzed include execution time, allocated memory, CPU usage, the space occupied by the models, and the ease of use of each framework. In addition, the accuracy of the final model was evaluated. Each of these evaluations is detailed below.

5.1 Execution Times

A standardized method was used for all operations to measure execution time consistently. The code implemented loaded the models, prepared the input data, performed the inferences, and measured the execution time for each inference. Each model was run 1000 times to ensure the robustness of the measurements.

During data collection, several time values were found that deviated significantly from the expected value. These outliers are attributed to interference on the device during execution, such as communication with servers and other activities that cannot be controlled. These anomalous values had much longer execution times than the rest of the values collected. To mitigate the impact of these outliers, the Interquartile Range (R_{IQ}) technique was used to filter them out.

The IQR method involves calculating the first quartile (Q_1) and third quartile (Q_3) of the execution times collected. The difference between Q_3 and Q_1 is the interquartile range. Values outside the range defined by $\mathrm{Q}_1 - 1.5\,R_{\mathrm{IQ}}$ and $Q_3 + 1.5\,R_{\mathrm{IQ}}$ were considered outliers and removed from the data set.

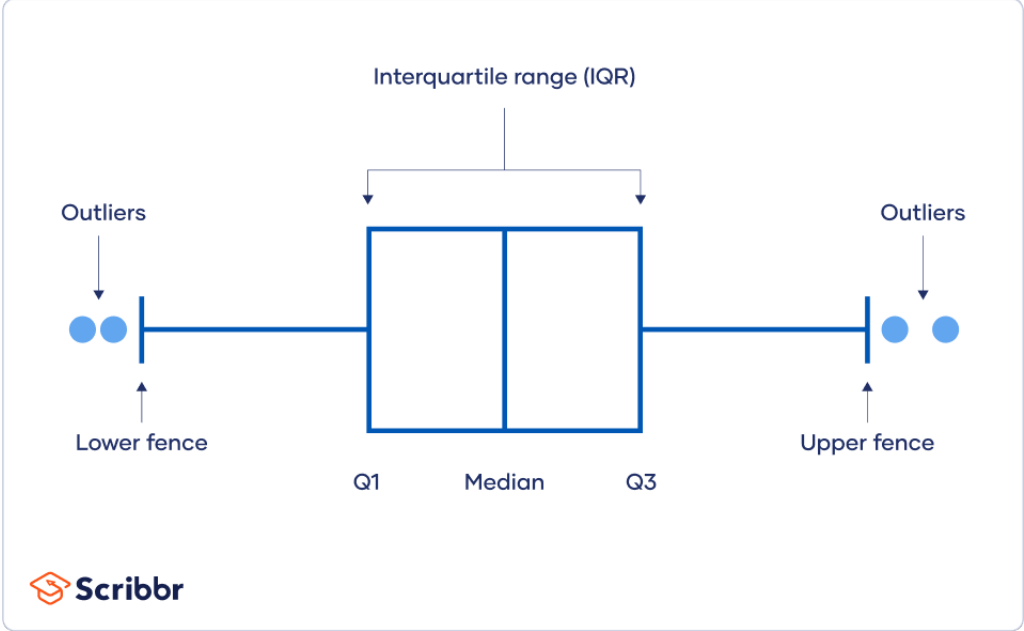
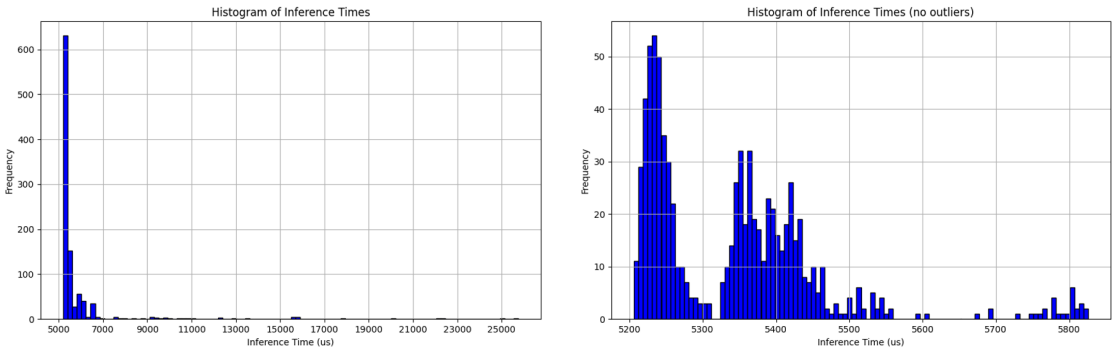


Figure 5.1: Box diagram showing the interquartile range (IQR), which is the statistical measure between the first quartile (Q1) and the third quartile (Q3). The centre line represents the median of the data. The limits are calculated as 1.5 times the IQR above the Q3 (upper fence) and below the Q1 (lower fence). Points beyond these limits are considered outliers. (from [25])

After removing the outliers, the remaining execution times were used to calculate the mean, standard deviation, and other relevant statistics. This process ensured that the execution times analyzed were representative and less subject to random variations caused by external factors.

The graphs of Fig. 5.2 are an example of the results of a model used during these experiments before and after removing the outliers:



(a) Before Outliers removal

(b) After Outliers removal

Figure 5.2: Model example with and without Outliers

5.1.1 ADD and SQRT Operations execution Times

Analyzing the results of the execution times in the following table 5.1 for the ADD and SQRT operations using the different frameworks and conversion methods, it is possible to notice some interesting patterns.

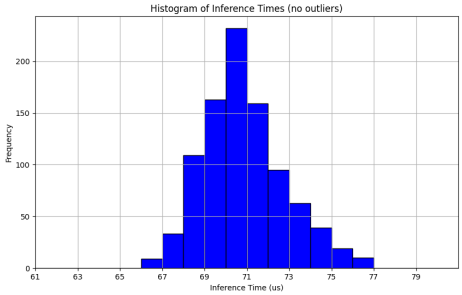
Table 5.1: Execution times for ADD and SQRT operations across different frameworks and conversions

Operation	**Framework/ Conversion**	**Average executiontime (μs)**	**Standard deviation (μs) %**	**Number of outliers**	**Minimum executiontime (μs)**
ADD	TFLite	70.35	1.93 (2.75%)	68	66
ADD	TensorFlow to ONNX	242.07	6.66 (2.75%)	89	225
ADD	Pytorch to ONNX Export	248.19	6.52 (2.63%)	100	231
ADD	Pytorch to ONNX Dynamo	347.08	8.16 (2.35%)	101	326
SQRT	TFLite	362.56	2.92 (0.80%)	137	356
SQRT	TensorFlow to ONNX	297.20	6.51 (2.19%)	112	282
SQRT	Pytorch to ONNX Export	296.73	7.42 (2.50%)	104	282
SQRT	Pytorch to ONNX Dynamo	308.55	7.14 (2.31%)	135	289

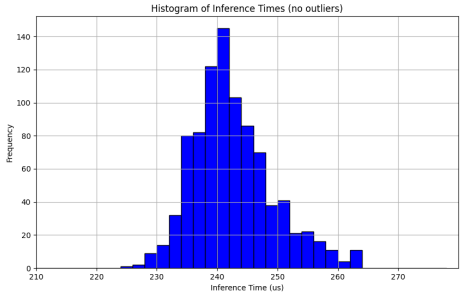
For the ADD operation, the results show that the TFLite framework performs significantly better compared to the various conversions to ONNX. The average executiontime using TFLite is approximately 70.35 µs, while the conversions from TensorFlow to ONNX and from PyTorch to ONNX (using both export and dynamo) show much longer average times, ranging from 242.07 µs to 347.08 µs. This difference can be explained by TFLite's optimized efficiency for simple operations such as ADD, which is specifically tuned for mobile and low-power devices. Conversions to ONNX, while useful for interoperability and scalability, introduce additional overhead due to the conversion process and the more generic nature of ONNX, which needs to handle a wide variety of operations and source frameworks.

However, when looking at the SQRT operation, the situation is a little different. Although TFLite still performs competitively with an average executiontime of 362,56 µs, the difference to ONNX conversions is not as pronounced. Conversions from TensorFlow to ONNX and from PyTorch to ONNX show average executiontimes ranging from 297,206 µs to 308,551 µs. This smaller deviation in performance between TFLite and ONNX can be attributed to the more complex nature of the SQRT operation compared to ADD. The SQRT operation may benefit more from the optimizations present in frameworks that convert to ONNX, especially when considering the underlying hardware implementation that these frameworks can leverage more efficiently for more complex mathematical operations.

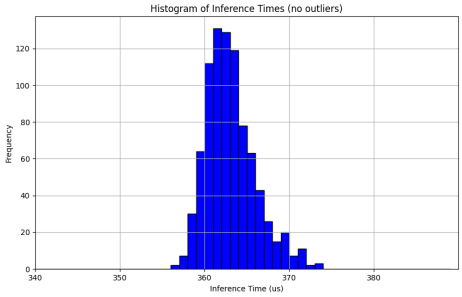
To get a better view of the executiontimes in these 1000 runs, histograms were generated for each case 5.3. Analyzing these histograms, it can be seen that the distribution of executiontimes for both operations follows an approximately normal or Gaussian distribution, with a slight asymmetry to the right. This asymmetry indicates the presence of some higher than average execution values, which in principle is due to temporal variations in hardware performance, resource management, and other systemic factors that introduce additional latency. This effect is more visible in more complex operations and models (which will be analyzed in more detail below), where memory management and the execution of several operations concurrently can cause more significant fluctuations in executiontimes.



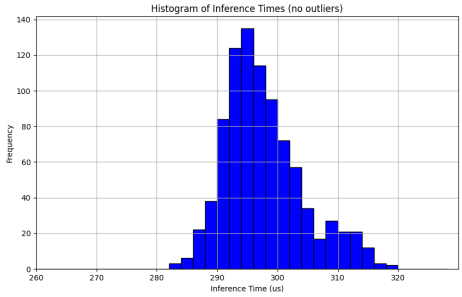
(a) TFLite ADD operation



(b) ONNX TensorFlow conversion ADD operation



(c) TFLite SQRT operation



(d) ONNX TensorFlow conversion SQRT operation

Figure 5.3: ADD and SQRT executiontime Histograms

In summary, the TFLite framework shows a clear advantage in terms of executiontimes for the ADD operation, reflecting its optimization for simple operations on low-power devices. For the SQRT operation, although TFLite continues to be efficient, conversions to ONNX show shorter execution times, highlighting how more complex operations can better balance the optimizations offered by different frameworks and conversion methods.

5.1.2 SORT & DFT operations Execution Times

As in the previous Section 5.1.1 for the ADD and SQRT operations, a detailed analysis of execution times was also carried out for the SORT and DFT operations, using different frameworks and

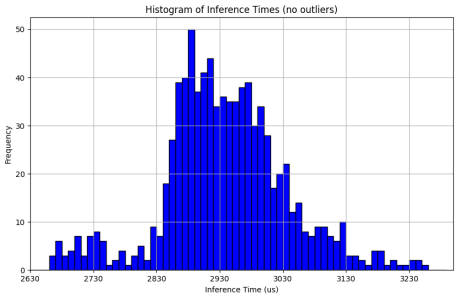
Table 5.2: Execution times for FFT and Sort operations across different frameworks and conversions

Operation	**Framework/ Conversion**	**Average execution time (μs)**	**Standard deviation (μs) %**	**Number of outliers**	**Minimum execution time (μs)**
FFT	TFLite	1426.05	46.55 (3.26%)	117	1368.33
DFT	ONNX Script	6042	91.83 (1.51%)	138	5920.47
DFT	Pytorch Dynamo	8446.09	393.8 (4.66%)	162	8100.83
Sort	TFLite	2511.24	52.2598 (2.08%)	97	2466.77
Sort	TFLite to ONNX conversion	4892.66	88.38 (1.85%)	132	4786.94
Sort	Pytorch to ONNX export conversion	5196.69	93.40 (2.28%)	151	5130.57
Sort	TopK Pytorch to ONNX dynamo conversion	5340.88	122.06 (2.28%)	169	5308.91

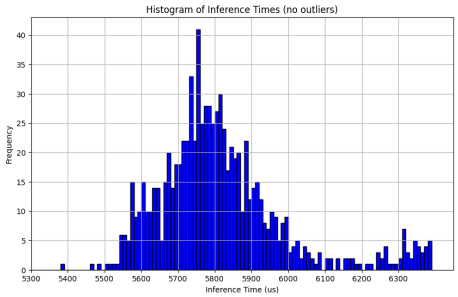
conversion methods. The data is presented in the table below 5.2, where the average execution times, standard deviations, number of outliers and minimum execution times are listed.

For the SORT operation, TFLite has the shortest average execution time, with 2511.24 μs, followed by the conversion from TFLite to ONNX, with 4892.66 μs. Conversions from Pytorch to ONNX (both export and dynamo) have higher average execution times, at 5196.69 μs and 5340.88 μs, respectively. The SORT operation in TFLite, in addition to showing the lowest average time, also shows relatively low variability, with a standard deviation of 52.2598 μs (2.08%). In contrast, the conversion from Pytorch to ONNX dynamo shows the greatest variability, with a standard deviation of 122.06 μs (2.28%). As was also seen in the previous section with the simpler operations, it is possible to see the greater simplicity and optimization of the TFLite framework for execution operations on low-power devices, resulting in shorter and more consistent execution times.

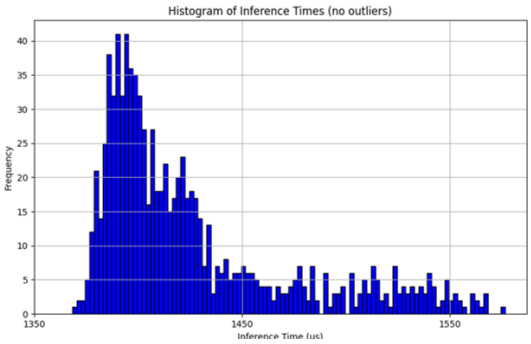
As for the DFT operation, the average execution times are significantly higher for all frameworks and conversion methods, reflecting the greater complexity of this operation. The lowest average execution time is presented by TFLite, with 1426.05 us, while both ONNX models exhibit a substantially higher average time, with 5920.47 μs and 8446.09 us. The variability in execution times for DFT is also more pronounced, especially in the Pytorch Dynamo conversion, which has a standard deviation of 393.8 μs (4.66%). This behavior was expected due to the absence of the FFT operation in the ONNX Framework. Furthermore, it is important to note that the DFT operation is intrinsically slower than the FFT, due to its more computationally intensive nature, as discussed in the 4.4.4 subsection.



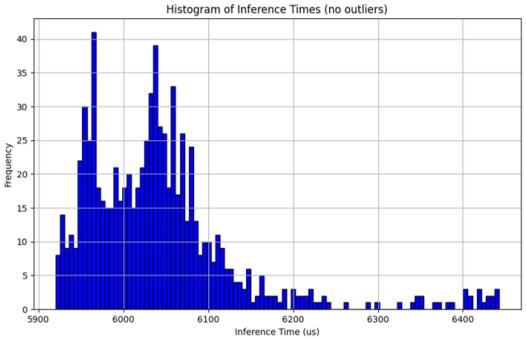
(a) TFLite SORT operation



(b) ONNX dynamo conversion SORT operation



(c) TFLite FFT operation



(d) ONNXScript DFT operation

Figure 5.4: DFT/FFT and SORT execution time Histograms

The histograms of the DFT and FFT operations 5.4 show a steep slope to the right, indicating that most of the execution times cluster in a lower range, with a tail extending to the right. This type of distribution is called a right-skewed distribution. This behavior has been observed before in simpler operations 5.1.1, but in this case it is much more pronounced and noticeable. This slope suggests variability in execution times, due to the heterogeneous hardware environment.

The reason for this "right-skewed" phenomenon being more noticeable in more complex operations, such as DFT and FFT, compared to simpler operations, lies in the complexity and variation introduced. More complex operations tend to introduce more variability in execution times due to the greater number of operations involved and the greater likelihood of external factors affecting performance, such as resource management and memory latencies. For simple operations, variability is minimal, resulting in distributions that are closer to a normal distribution. However, the variability is greater for complex operations, resulting in more asymmetrical and scattered distributions.

In addition, a certain rightward tilt is also observed in the SORT operation, although it is less significant. This suggests that, although there are some variations in the execution times, the SORT operation generally shows greater consistency. The smaller skew to the right in SORT operations can be justified by algorithmic complexity: while DFT has a complexity of $O(N^2)$, many sorting algorithms, such as quicksort and mergesort, are deterministic and have well-defined time complexities of $O(N \log N)$. This results in fewer discrepancies in the execution conditions, possibly

due to better optimization or more stable handling of the calculations in the SORT operation.

In addition to the skew to the right, the distributions of the execution times for the DFT operations using ONNXScript and ONNX Dynamo show bimodal characteristics. The presence of two distinct peaks in the histograms suggests that different factors or conditions can influence execution times. This bimodality may indicate two types of execution scenarios: one where the operation is performed optimally and another where certain overheads or delays are introduced. These can be due to variations in system load, memory availability, or differences in the way the hardware handles certain computational patterns.

To deepen the analysis and confirm that the hardware influences the observed distributions, additional tests were carried out using a 64-bit x86-based system. The SORT model was run on this machine, and the results showed a distribution much closer to normal, as illustrated in Figure 5.5.

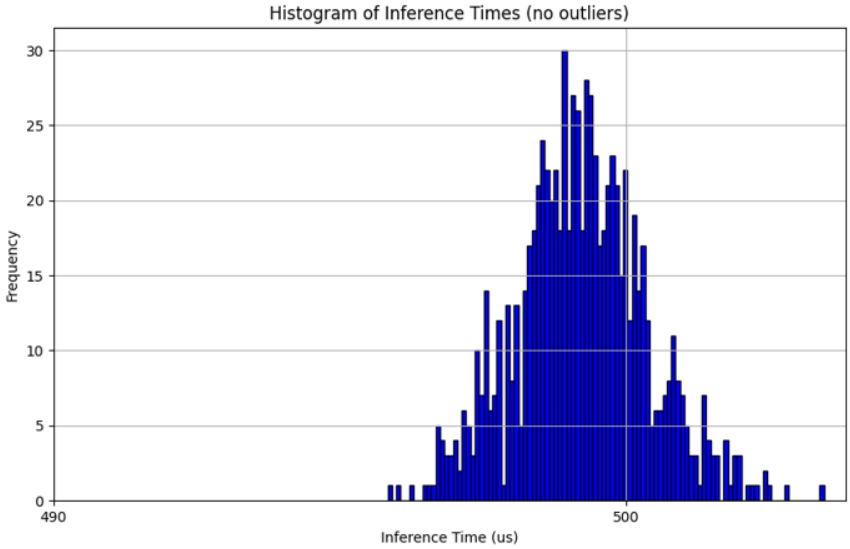


Figure 5.5: Histogram of Execution Times on 64-bit x86 Computer

Analysing the execution times on the 64-bit x86 computer showed a normal distribution, different from the asymmetry observed in the tests with low-power hardware devices. This difference can be attributed to the greater stability and more effective resource management of more powerful hardware, such as the x86. These results suggest that the variability observed in the distributions of execution times on lower-capacity devices is strongly associated with the limitations and specific characteristics of the hardware used.

Although the hardware differences are evident, it can be seen that TFLite performs the best optimization when compared to the ONNX distributions. When examining the histograms of the SORT operations, it is clear that TFLite shows a more concentrated and less dispersed distribution compared to ONNX, which shows greater variability and a wider distribution of execution times.

Table 5.3: Execution times for ONNX and TFLite loop models

Model	**Framework/ Conversion**	**Average Execution Time (ms)**	**Minimum Execution Time (ms)**	**Standard Deviation Time (ms)**	**Number of Outliers**
Loop	TFLite	2924.91	2805	77.46	0
Loop	TensorFlow to ONNX conversion	4213.25	4051	64.03	4

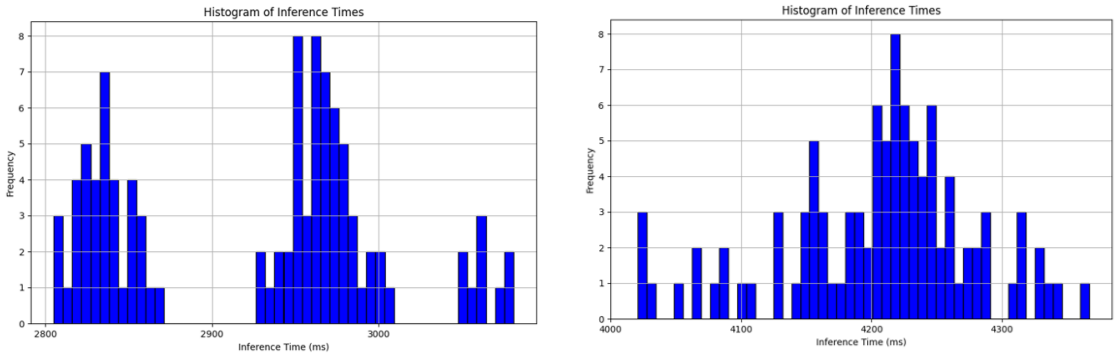
This behavior is consistent in both operations, indicating that TFLite manages to maintain greater consistency and optimization.

5.1.3 LOOP model Execution Times

For the LOOP model, a comparative analysis was carried out between the distributions of execution times using TFLite and ONNX. As this model is substantially larger than the previous ones, as seen earlier in the study of the models in Sec. 4.4.5, the execution times for each run are considerably longer. For this reason, only 100 runs were carried out instead of the 1000 used in the smaller models. Table 5.4 shows the execution times for the LOOP model:

By analyzing Table 5.4, it can be seen that TFLite continues to be better optimized than ONNX, both in terms of average execution time and consistency of execution times. TFLite shows an average execution time of 2924.91 ms, with a standard deviation of 77.46 ms (2.65 %), while ONNX shows an average time of 4213.25 ms and a standard deviation of 64.03 ms (1.52 %). In addition, TFLite had no outliers, while ONNX had 4 outliers.

An interesting fact when analyzing the histograms 5.6 is the presence of multiple peaks, especially in the TFLite histogram. This variation can be attributed to different factors. The complexity of the LOOP model is significantly higher, which can result in variability in execution time as different parts of the model are processed. Different runs may involve different execution paths within the LOOP model, resulting in variations in the time taken to complete each execution. This is especially likely in complex models that contain several conditional operations and loops. Hardware resource management during execution can introduce variations, as different executions can



(a) TFLite LOOP operation

(b) ONNX Tensorflow conversion Loop model

Figure 5.6: Loop model Execution time Histograms

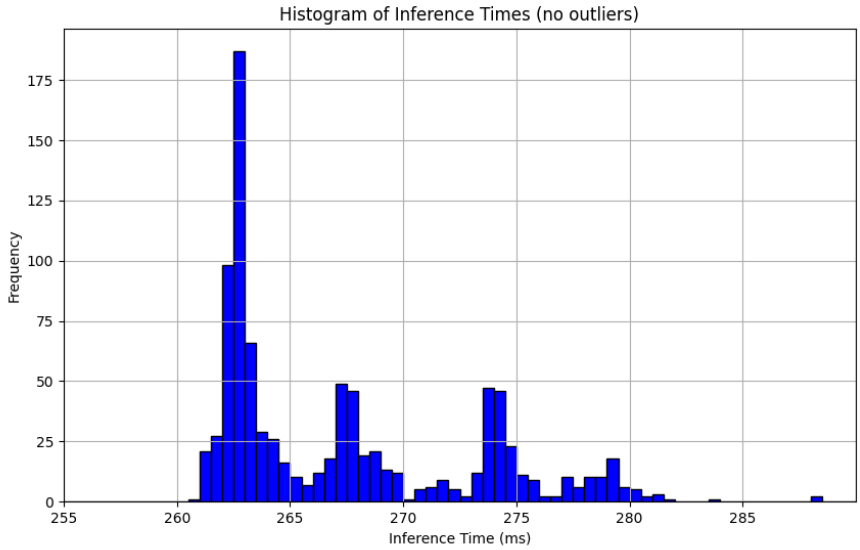


Figure 5.7: Histogram of Execution Times of the Collision Detection Model on TFLite Framework

result in different memory usage patterns and available processing units, leading to variations in execution time.

It can be seen that the more complex the model, the greater the variability in execution times. In the simplest operations, the distribution was normal, as seen in the ADD and SQRT models. The distribution shifted to a bimodal form when the complexity was increased with the SORT and DFT models. Now, with an even larger model such as LOOP, there are already three distinct peaks in the histograms. These multiple peaks form a multimodal distribution, representing different forms of model execution. This reflects the complex and variable nature of running the LOOP model, where the different peaks represent the various ways in which the model has been inferred.

5.1.4 Collision Detection Model Execution Times

In this section, is presented the results of the execution times for the Bosch collision detection model implemented in the TensorFlow Lite (TFLite) framework. The histogram shown represents the distribution of execution times (in milliseconds) without considering outliers. The data clearly illustrates the points discussed in previous sections: as the complexity of the model increases, there is a tendency for multiple peaks in the execution times. This phenomenon can be attributed to several factors, including the variety of operations and layers involved in the model, which may lead to different execution paths and resource usage patterns.

The histogram reveals several distinct peaks, indicating that the execution times are not uniformly distributed. Instead, they cluster around specific values. This behavior can be explained by the model's complexity, the nature of the data being processed, and the variability inherent in the execution environment. The largest cluster of execution times occurs within the 260-265 ms

Table 5.4: Execution times for TFLite collision detection model

Model	**Framework/ Conversion**	**Average Execution Time (ms)**	**Minimum Execution Time (ms)**	**Standard Deviation Time (ms)**	**Number of Outliers**
Loop	TFLite	267.03	261.01	5.725 (2.14%)	66

range, suggesting that the majority of executions are completed quickly and efficiently. This peak likely represents the baseline performance of the model under typical conditions.

Additional clusters appear around 265-275 ms, which may correspond to more complex scenarios where the model requires additional computation. These peaks indicate the presence of variations in the data or specific model operations that introduce latency. Less frequent but still significant peaks occur in the 275-285 ms range, representing the tail end of the distribution where the most complex or resource-intensive executions occur.

Another crucial factor contributing to the observed variability in execution times is the execution environment itself. On devices where multiple processes are running concurrently, background tasks can influence the availability of computational resources. This can lead to fluctuations in the execution times as the model competes for CPU and memory resources with other applications. Such variability is particularly pronounced in resource-constrained environments where the hardware is being utilized to its full capacity.

The observed distribution of execution times highlights the challenges of deploying complex models in resource-constrained environments. The multiple peaks suggest that the model's performance can vary significantly depending on the input data, specific operations being executed, and the state of the device at any given time. This variability must be accounted for when designing systems that rely on real-time processing and decision-making. For applications that require consistent and predictable response times, the presence of multiple peaks in the execution time distribution can be problematic. Systems must be designed to handle these variations, possibly by incorporating buffering or prioritization mechanisms.

In conclusion, while the Bosch collision detection model shows promising performance in the TFLite framework, its complexity introduces variability in execution times. Addressing this variability, including the impact of concurrent processes and the state of the execution environment, is essential for the successful deployment of the model in real-world applications where consistent and efficient performance is critical.

5.2 Memory

Efficient memory management is a crucial factor when implementing machine learning models, especially on devices with limited resources. In this section, the memory utilization of the various frameworks tested is evaluated.

5.2.1 Size of Executables and Libraries

The size of executables and libraries is an important factor to consider. TensorFlow Lite uses static linking, incorporating all the necessary dependencies within the executable itself. This results in larger files but simplifies the distribution and execution of applications on devices where additional libraries cannot be installed.

On the other hand, ONNX Runtime uses dynamic libraries, such as `libonnxruntime.so.1.18.0`, keeping the size of executables smaller and allowing libraries to be shared between multiple programs. This approach is advantageous in terms of global memory usage when using multiple executables and the flexibility to update the libraries without the need to recompile the executables.

Table 5.5: Executable size comparison of the executables and libraries

Framework	**Component**	**Size (KB)**
ONNX Runtime	Executable	123.540
	Dynamic Library	7361.944
	Total (executable + library)	**7485.484**
TensorFlow Lite	**Total(Executable)**	**3810.960**

5.2.2 Allocated Memory During Execution

During the memory analysis of the frameworks, memory benchmarks were carried out at critical allocation points to understand memory consumption at different stages of the process. The SORT model was used as an example, but the same behaviour was observed for other models such as ADD, SQRT and LOOP, only with different values, which will be analysed later.

In TensorFlow Lite, the initial memory recorded before any code execution is 1300 KB for any model. The executable uses this initial memory and includes the basic overhead of the operating system, the execution of the binary, the allocation of the initial heap, and any libraries loaded by the C++ runtime. After creating the interpreter with the line, the memory recorded was 1960 KB. This increase in memory is due to the loading of the model and the creation of the interpreter, which involves memory allocations to store the model itself, TensorFlow Lite's internal data structures, and other initialization overheads.

After allocating the tensors with `TfLiteInterpreterAllocateTensors();`, the memory registered was 1992 KB. Allocating the tensors prepares the memory buffers needed for the model's inputs and outputs. This step may include allocating additional memory to store the tensors and any auxiliary structures needed for eecution. Obtaining the input tensor with `TfLiteInterpreterGetInputTensor()` did not significantly alter the memory, which remained at 1992 KB. Before execution, the memory recorded was 2060 KB. Filling the input tensor with data can cause small additional allocations, especially if there are internal operations to configure the data. After execution, the memory registered was 2124 KB. Running execution can

involve internal temporary allocations used during model processing. TensorFlow Lite can create additional internal buffers to manage the data during execution. The incremental increase in memory during subsequent executions may have been due to additional memory management overheads or small dynamic allocations that are not completely released between executions.

In the case of ONNX Runtime, the initial memory recorded before any code execution is 1456 KB for any model. As with TensorFlow Lite, this initial memory includes the basic overhead of the operating system, the execution of the binary, the initial heap allocation, and any libraries loaded by the C++ runtime. Before loading the model with `Ort::Session session();`, the memory registered was 1684 KB. Initializing the ONNX environment and session options allocates additional memory needed for the ONNX Runtime's internal structures.

After loading the model, the memory registered was 4256 KB. Loading the ONNX model involves reading the model file and allocating memory to store all the layers, weights, and other structures needed for execution. Before creating the CPU, the memory registered was 4320 KB. This stage did not involve significant memory changes, as it only prepared for the creation of memory information. Before allocating the tensors, the registered memory remained at 4320 KB, indicating that the memory needed for the input data had been prepared, but the actual memory had not yet been allocated. Before execution, the registered memory was 4320 KB. Preparing for execution may not add significant additional memory, just setting up the data. After the execution, the registered memory was 4692 KB. Running the execution involves internal temporary allocations used during model processing. ONNX Runtime can create additional internal buffers to manage the data during execution. The continuous increase in memory during subsequent executions may have been attributable to the use of shared dynamic libraries and ONNX Runtime's internal memory management, which may not completely free up all the allocated memory between executions.

5.2.3 Comparison of Total Allocated Memory

A comparison of the total memory allocated by the ADD, SQRT, SORT and LOOP models in different frameworks shows a clear difference in the efficiency of memory use. In TensorFlow Lite, total memory usage stabilises quickly after the first execution and remains constant. The total memory allocated is significantly lower compared to ONNX Runtime. For the same models, the memory allocated by ONNX Runtime continues to increase over multiple executions, resulting in higher total memory usage, but offering greater flexibility for different types of executions.

For example, in the SORT model, the memory allocated in TensorFlow Lite stabilizes quickly after the first execution, showing little subsequent variation, due to the use of static links and FlatBuffers mentioned earlier in the study of frameworks (see Section 4.2.1), which allow for efficient memory allocation and direct data access. In contrast, in the ONNX Runtime, memory increases progressively with each execution, starting at 1456 KB and increasing up to 5724 KB, due to dynamic memory allocation and the use of Protocol Buffers 4.2.3, which introduce an additional parsing and memory management overhead. In the LOOP model, the behavior is similar, with TensorFlow Lite showing a rapid stabilization in memory thanks to static linking and the use of FlatBuffers, while ONNX Runtime shows a progressive increase in memory, starting at 1456 KB

Table 5.6: Total allocated memory for ONNX and TFLite models and respective model sizes

Model	Framework/ Conversion	Total Allocated Memory (KB)	Model Size (Bytes)
ADD	TFLite	2132	780
	ONNX (TensorFlow)	4828	177
	ONNX (Torch Export)	4892	124
	ONNX (Dynamo)	5132	1025
SQRT	TFLite	2116	780
	ONNX (TensorFlow)	4856	176
	ONNX (Export)	4824	115
	ONNX (Dynamo)	5236	806
SORT	TFLite	2236	1236
	ONNX (TensorFlow)	5724	360
	ONNX (Export)	5136	862
	ONNX (Dynamo)	5720	2530
DFT	TFLite	2236	1192
	ONNX (Pytorch)	5128	3344
	ONNX (Script)	5772	326
LOOP	TFLite	2176	6400
	ONNX (TensorFlow)	4848	10514
Collision Detection	TFLite	14692	8198400

and reaching 4848 KB after several executions, again due to the use of dynamic memory allocation and Protocol Buffers.

These differences reflect different approaches to implementing machine learning models, each with its advantages and disadvantages depending on the context of use. The choice between static and dynamic linking must take into account the specific requirements of portability, resource utilization, and simplicity of distribution for each application. TensorFlow Lite, with its static allocation approach and use of FlatBuffers, proves to be more efficient in terms of memory usage, making it an excellent choice for devices with limited resources. On the other hand, ONNX Runtime, with its dynamic allocation flexibility and use of Protocol Buffers, may be more suitable for scenarios where flexibility and adaptability are more important.

5.2.4 Comparison of Total Model Size

In this section, the total sizes of different models converted to ONNX and TFLite formats are compared. It is important to note that simpler models such as ADD and SQRT tend to have smaller sizes when converted to ONNX compared to TFLite. For instance, the ADD model size in ONNX is 177 Bytes (TensorFlow Conversion) and 124 Bytes (Torch Export conversion) versus 780 Bytes in TFLite. Similarly, the SQRT model size in ONNX is 176 Bytes (TensorFlow) and 115 Bytes (Torch Export) compared to 780 Bytes in TFLite. This reduction in size for simpler models can be advantageous in terms of memory efficiency and faster loading times.

However, this trend does not hold for more complex models. For example, the LOOP model, which is significantly more complex, has a larger size in ONNX compared to TFLite. Specifically, the LOOP model size in ONNX (TensorFlow conversion) is 10514 Bytes, while in TFLite it is 6400 Bytes. This indicates that as the complexity of the model increases, ONNX models can become larger, potentially affecting memory usage and loading times.

Additionally, it is evident that the complexity introduced by the Dynamo Export conversion method as said in other secctions results in larger model sizes. The ADD model size using Dynamo Export is 1025 Bytes, which is considerably larger than the sizes obtained through other conversion methods. This pattern is also observed in other models, suggesting that the Dynamo Export method, while potentially offering other benefits, tends to increase the model size significantly.

Overall, this comparison emphasises the importance of considering the complexity of the model and the conversion method chosen when evaluating memory usage. While ONNX can offer smaller sizes for simpler models, TFLite can be more efficient for complex models. In addition, the choice of conversion method, such as Dynamo Export, can significantly impact the size of the model and should be carefully considered based on the specific requirements and limitations of the implementation environment.

Following this logic, although it was not possible to convert the collision detection model, based on the observations made here, it can be estimated that the model would be larger in ONNX than in TFLite.

5.3 Ease of Use and Support

When deploying machine learning models in various environments, ease of use and support are critical factors to consider. This section evaluates the usability and support infrastructure of the primary frameworks utilized in this study: TensorFlow Lite, ONNX Runtime, and PyTorch. Additionally, this evaluation includes subjective insights from my own experience with these frameworks.

5.3.1 Evaluating TensorFlow Lite Usability and Support

TensorFlow Lite is designed with a strong focus on ease of use, particularly for mobile and embedded devices. Its ecosystem includes comprehensive documentation, tutorials, and examples that guide users through the process of converting, optimizing, and deploying models. TensorFlow Lite Converter simplifies the process of converting TensorFlow models into a format optimized for mobile and embedded environments. In addition, TensorFlow Lite Interpreter enables efficient model execution with minimal latency.

As for the framework's support, TensorFlow Lite enjoys broad support from a large community of developers and collaborators. Google provides official support through detailed documentation, forums, and a dedicated repository on GitHub. Regular updates and active community participation ensure that users have access to the latest features and bug fixes.

In my experience, TensorFlow Lite offers a seamless development process, from model conversion to deployment. The abundant resources and active community support have been invaluable, making it relatively simple to troubleshoot and find solutions. It was the framework I most enjoyed developing the models, as I found it to have the most flexibility and breadth of operations and functions.

5.3.2 Evaluating ONNXRuntime Usability and Support

ONNX Runtime aims to provide a flexible, high-performance engine for running ONNX models on various platforms. Its design prioritizes interoperability, allowing models trained in different frameworks to be deployed seamlessly. The process of converting models to ONNX format is well documented, and the runtime itself is easy to integrate into both existing and new applications. However, users may encounter a steeper learning curve when dealing with the complexities of model conversion and optimization.

Supported by Microsoft, ONNX Runtime benefits from robust official documentation and a growing community. The framework receives regular updates, and the repository on GitHub serves as a central hub for tracking issues and community contributions. Microsoft also provides additional resources, such as webinars, tutorials, and technical support via forums and official channels.

In my experience, although the ONNX Runtime is highly versatile, I found the process of converting other frameworks to the ONNX format challenging at times, especially for complex models. In addition, the cross-compilation process for the device wasn't the easiest either. However, once mastered, its performance and flexibility make it a powerful tool for a variety of deployment scenarios.

In addition, I've had the opportunity to work with ONNXScript, and my experience wasn't the best. Compatibility with operations was difficult, and the interface for creating models was not very user-friendly. The interface, referring to the code needed to define models, is quite verbose and complex. This is due to the fact that ONNXScript is still in beta and under development. In addition, the support is also not ideal due to the lack of examples, since it is an early-stage technology.

5.3.3 Evaluating Pytorch Usability and Support

PyTorch is widely recognized for its user-friendly interface and dynamic computational graph, which makes it an excellent choice for research and development. Although the PyTorch deployment tool was not used in this project, I had the opportunity to use the framework and produce PyTorch models. The interface is very intuitive, with a vast amount of examples and support available, which makes it very easy to develop and deploy models.

Support for PyTorch is robust, reinforced by a vibrant community and support from Facebook. Comprehensive documentation, community forums, and many third-party tutorials and courses are

readily available. Regular updates and an active GitHub repository help maintain the framework's reliability and feature set.

In my experience, I've really enjoyed working with PyTorch due to its intuitive interface and the extensive support it offers. However, a significant drawback is that PyTorch models cannot be visualized by the visualization tool used in this project, Netron, which represents a limitation. This inability to visualise the models directly can make it difficult to analyse and verify the models developed, highlighting an area where the framework could improve.

5.4 Discussion of the results

After evaluating the different machine learning frameworks to determine the best option for deploying ML models in IoT applications in the mobility sector, can be concluded which framework is the most suitable for our specific case. The frameworks analyzed were TensorFlow Lite, ONNXRuntime, and PyTorch, each with its own unique strengths and weaknesses. This discussion aims to synthesize these findings to identify the most appropriate framework.

Starting with TensorFlow Lite, this framework stood out for its efficiency and optimization for mobile and embedded devices. Its low latency and reduced binary size guarantee highly efficient performance, which is essential for environments with limited resources, such as IoT devices. In addition, compatibility with a wide range of hardware platforms, including ARM and x86 architectures, as well as support for GPU acceleration, makes it an extremely versatile option. Ease of use is another strong point, thanks to tools such as TensorFlow Lite Model Maker, which simplify model conversion and optimization. Comprehensive documentation and extensive community support provide a solid foundation for developers, facilitating integration and troubleshooting.

On the other hand, ONNXRuntime stood out for its flexibility and ability to operate efficiently on various hardware configurations. Developed by Microsoft, ONNXRuntime optimizes the performance of models on different platforms, making it highly relevant for AIoT applications. Its ability to act as a universal intermediary for converting models between different frameworks adds significant flexibility, enabling interoperability between various machine learning tools. This attribute makes ONNX particularly useful when used as a framework converter. For example, in a team where some people are proficient in PyTorch and others in TensorFlow Lite, both can generate ONNX models to facilitate the deployment process, without having to use the original framework's interface directly. This can significantly simplify the workflow and integration of models in a multidisciplinary team. However, this flexibility comes with the disadvantage of longer execution and memory times. Even so, ONNXRuntime is a constantly evolving framework, and it is possible that, in the future, these trade-offs will become less noticeable, further improving its efficiency and applicability. Although it has robust support from Microsoft, the community and resources available may not be as extensive as those of other frameworks, mainly because some of its tools are still in the development process.

PyTorch, for its part, is widely recognized for its intuitive and flexible interface, dynamic computer graphics, and easy debugging capabilities, making it an excellent choice for prototyping and

experimentation. The support of an active community, extensive documentation, and continuous development driven by community feedback further strengthens its position. However, PyTorch is not primarily optimized for mobile and embedded devices, which can make its deployment on edge devices less efficient compared to TensorFlow Lite. Resource usage can be more intensive, representing a limitation for devices with memory and computing power constraints. Although PyTorch was not used as a deployment tool due to limitations imposed by the company, its use for converting models to ONNX format was explored. However, some limitations arose, especially with models that used "graph breaks" 4.4.5, which complicated the conversion. Even so, PyTorch's interface was easy to use, making it easy to prepare models before conversion.

After a detailed evaluation of each framework, it's concluded that TensorFlow Lite is the best overall choice for deploying ML models in IoT applications in the mobility sector. Its optimization for edge devices, ease of use, and robust support infrastructure make it the most suitable option for our specific case. However, ONNXRuntime offers an excellent alternative for scenarios that require high flexibility and cross-platform compatibility. PyTorch remains invaluable for research and development phases due to its user-friendly nature and robust community support.

The main objective of this research was to identify and evaluate ML frameworks that facilitate efficient and effective deployment in IoT applications for mobility services. The objective was to ensure that the selected framework can offer robust IoT solutions to improve passenger safety and comfort by focusing on criteria such as processing speed, memory utilization, ease of use, flexibility, and model accuracy. The results indicate that TensorFlow Lite best meets these objectives, offering a balanced approach between performance, compatibility, and usability.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This research provided a comprehensive analysis of machine learning (ML) frameworks for deployment in resource-constrained environments, with a specific focus on IoT applications in the mobility sector. The primary objective was to evaluate TensorFlow Lite (TFLite), ONNXRuntime, and PyTorch to determine their efficiency and suitability for real-world applications.

Through extensive experimentation and benchmarking, it was observed that TensorFlow Lite consistently offered the best balance between performance, memory usage, and ease of deployment. Its optimization for edge devices and robust support infrastructure make it particularly well-suited for deploying ML models in environments with limited computational resources.

ONNXRuntime demonstrated excellent cross-platform compatibility and flexibility, making it a viable alternative for scenarios requiring high interoperability between different systems. However, it was noted that the conversion process from TensorFlow to ONNX faced significant challenges, especially with complex models like the collision detection model. These challenges highlight the importance of choosing the right conversion tools and methods based on the specific requirements of the deployment environment.

PyTorch, while not used as a deployment tool in this study due to organizational constraints, proved invaluable for model preparation and conversion to ONNX format. Its user-friendly interface and strong community support make it an excellent choice for the research and development phase.

The study of the Bosch collision detection model, converted to TensorFlow Lite, highlighted the framework's ability to handle complex models efficiently. The histogram of inference times demonstrated the variability in performance, with distinct peaks indicating different computational paths and resource usage. This variability underscores the need for careful optimization and resource management in real-time applications.

In conclusion, while TensorFlow Lite emerged as the most suitable framework for this study, ONNXRuntime and PyTorch offer valuable capabilities that can be leveraged based on specific needs. The inability to convert the collision detection model to ONNX, coupled with the observed

patterns in model size and performance, suggests that TensorFlow Lite's efficiency is crucial for complex models in resource-constrained environments. Future work should continue to explore and refine these frameworks to enhance the deployment of ML models in IoT applications, ensuring optimal performance, compatibility, and usability.

6.2 Future Work

During the course of this research, a new machine learning framework was discovered at a more advanced stage of the project. Due to time constraints, it was not feasible to include a comprehensive study of this framework within the scope of this dissertation. Future work should aim to evaluate this newly discovered framework, focusing on its compatibility, computational efficiency, and deployment ease on heterogeneous hardware platforms used in IoT systems.

In addition, expanding on the optimization techniques explored in this dissertation could further enhance the performance of machine learning models on resource-constrained devices. Techniques such as model pruning and advanced quantization could provide additional improvements. Integrating newer hardware acceleration technologies that have emerged since the completion of this project could also yield significant benefits in terms of inference times and energy efficiency.

Glow performs sophisticated optimizations such as operator fusion, where multiple neural network operations are combined into a single operation, reducing memory bandwidth and increasing execution speed. By compiling code tailored to the target hardware, Glow can leverage specific processor features, such as SIMD (Single Instruction, Multiple Data) instructions and specialized accelerators, to maximize performance. The modular design of Glow allows for easy extension and integration with new hardware targets and custom operators, making it adaptable to evolving AI and hardware landscapes.

Integrating Glow into the existing framework could address some of the current limitations related to energy consumption and model inference speed. Future research should evaluate the potential of Glow in optimizing machine learning models for AIoT applications, specifically in the context of intelligent mobility and resource-constrained environments. Investigating the capabilities of Glow aims to enhance the scalability, efficiency, and adaptability of machine learning deployments in IoT and edge computing scenarios. This exploration could lead to significant improvements in the performance and sustainability of AI-driven mobility solutions.

Including Glow in future research opens new avenues for optimizing machine learning frameworks, potentially revolutionizing the deployment of AI models in resource-limited environments.

References

[1] Apache TVM Documentation, March 2024. Available at https://tvm.apache.org/docs/. Cited on pages 9 and 21.

[2] ONNX Runtime, March 2024. Available at https://onnxruntime.ai/docs/. Cited on pages 8, 9, and 21.

[3] ONNXscript, April 2024. Available at https://onnxscript.ai/. Cited on page 24.

[4] Hyunho Ahn, Tian Chen, Nawras Alnaasan, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, Dhabaleswar K., and Panda. Performance characterization of using quantization for dnn inference on edge devices: Extended version, 2023. Available at https://arxiv.org/abs/2303.05016. Cited on page 17.

[5] Microsoft Azure. How to accelerate DevOps with Machine Learning lifecycle management, 7 2023. Available at https://azure.microsoft.com/en-us/blog/how-to-accelerate-devops-with-machine-learning-lifecycle-management/. Cited on pages x and 12.

[6] João Paulo Bedretchuk, Sergio Arribas Garcia, Thiago Nogiri Nogiri Igarashi, Rafael Canal, Anderson Wedderhoff Spengler, and Giovani Gracioli. Low-cost data acquisition system for automotive electronic control units. *Sensors*, 2023. Cited on page 14.

[7] Capitalising on the benefits of Artificial Intelligence of Things: Opportunities presented by AIoT, February 2024. Available at https://www.credera.co.uk/insights/capitalising-the-artificial-intelligence-of-things. Cited on pages x and 10.

[8] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu, and Hucheng Wang. A vision of IoT: Applications, challenges, and opportunities with china perspective. *Ieee Internet of Things Journal*, 2014. Cited on page 2.

[9] J. Clancy Clements and Yingjie Lao. Hardware trojan design on neural networks. 2019. Cited on page 1.

[10] Arm Deveoper. Documentation – Arm Developer, May 2024. Available at https://developer.arm.com/documentation/den0018/a/Compiling-NEON-Instructions/GCC-command-line-options/Option-to-specify-the-FPU. Cited on page 30.

[11] Róbert Dávid, Jared Duke, Abhinav Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv (Cornell University)*, 10 2020. Cited on page 7.

[12] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016. Cited on page 17.

[13] Jovan Ivković and Jelena Lužija Ivković. Analysis of the performance of the new generation of 32-bit microcontrollers for IoT and big data application. In *Proceedings of the Conference on IoT and Big Data Applications*. ResearchGate, March 2017. Conference Paper. Cited on pages 18 and 19.

[14] TensorFlow Lite. TensorFlow Lite | ML for mobile and edge devices, April 2024. Available at https://www.tensorflow.org/lite. Cited on pages 7 and 21.

[15] Arm Ltd. Downloads | GNU-A Downloads – Arm Developer, March 2024. Available at https://developer.arm.com/downloads/-/gnu-a. Cited on page 30.

[16] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. Comparison and benchmarking of ai models and frameworks on mobile devices, 2020. Cited on pages 15 and 16.

[17] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Comput. Surv.*, 54(8), oct 2021. Cited on page 21.

[18] Netron. Netron, February 2024. Available at https://netron.app/. Cited on pages 33 and 34.

[19] ONNXruntime. ORT model format runtime optimization, March 2024. Available at https://onnxruntime.ai/docs/performance/model-optimizations/ort-format-model-runtime-optimization.html. Cited on page 32.

[20] Kieu-Ha Phung, Hieu Tran, Thang Nguyen, Hung V. Dao, Vinh Tran-Quang, Thu-Huong Truong, An Braeken, and Kris Steenhaut. oneVFC—A vehicular fog computation platform for artificial intelligence in internet of vehicles. *IEEE Access*, 9:117456–117470, 2021. Cited on page 14.

[21] Evgeny Ponomarev, Sergey Matveev, Ivan Oseledets, and Valery Glukhov. Latency estimation tool and investigation of neural networks inference on mobile GPU. *Computers*, 10(8), 2021. Cited on page 16.

[22] Pytorch. PyTorch documentation — PyTorch 2.1 documentation, April 2024. Available at https://pytorch.org/docs/stable/index.html. Cited on pages 8 and 21.

[23] PyTorch. torch.export Tutorial — PyTorch Tutorials 2.3.0 documentation, May 2024. Available at https://pytorch.org/tutorials/intermediate/torch_export_tutorial.html. Cited on page 46.

[24] ONNX Runtime. Build ONNX runtime with execution providers, March 2024. Available at https://onnxruntime.ai/docs/build/eps.html. Cited on page 32.

[25] Scribbr. How to identify and handle outliers in your data, May 2024. Accessed: 2024-06-17. Cited on page 56.

[26] Max Sponner, Bernd Waschneck, and Akash Kumar. Compiler toolchains for deep learning workloads on embedded platforms. *arXiv preprint arXiv:2104.04576*, 2021. Cited on page 21.

[27] Evariste Twahirwa, James Rwigema, and Raja Datta. Design and deployment of vehicular internet of things for smart city applications. *Sustainability*, 2021. Cited on page 1.

[28] Analytics Vidhya. Speedup PyTorch inference with ONNX, May 2024. Available at https://medium.com/analytics-vidhya/speedup-pytorch-inference-with-onnx-284d6a47936e. Cited on pages x and 26.

[29] Shen Wang and Zhuobiao Qiao. Robust pervasive detection for adversarial samples of artificial intelligence in IoT environments. *IEEE Access*, 2019. Cited on page 1.

[30] What is an inertial measurement unit, February 2024. Available at https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu. Cited on page 13.

[31] Marek Żyliński, Amir Nassibi, Ildar Rakhmatulin, Adil Malik, Christos Michael Papavassiliou, and Danilo P. Mandic. Deployment of artificial intelligence models on edge devices: A tutorial brief. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 71(3):1738–1743, 2024. Cited on page 17.