

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Collaborative Machine Learning in Far-Edge and Edge Environments

Tomás Alexandre Soeiro Vicente



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Luís Almeida

Second Supervisor: Fernando Rego

June 30, 2026

Towards Collaborative Machine Learning in Far-Edge and Edge Environments

Tomás Alexandre Soeiro Vicente

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Rui Pinto

Examiner: Prof. Luís Moutinho

Member: Prof. Luís Almeida

June 30, 2026

Resumo

Num mundo onde a Inteligência Artificial (IA) está cada vez mais presente em aplicações do dia a dia, a estratégia para desenvolver e implementar modelos de IA no estado da arte tem dependido fortemente de servidores potentes hospedados na nuvem. No entanto, esta abordagem apresenta várias desvantagens, incluindo a distância entre a fonte de dados, o servidor e o utilizador final, o que pode levar ao aumento da latência, a um maior consumo de energia e a potenciais vulnerabilidades de privacidade. Para abordar estas questões, a computação de borda (edge computing) surgiu como uma solução promissora, aproximando os recursos computacionais de onde são necessários, em particular, através do paradigma TinyML, que oferece a possibilidade de executar modelos de IA em dispositivos de borda. No entanto, os dispositivos na borda extrema (far-edge), como os microcontroladores, embora altamente limitados em termos de memória e computação, são amplamente utilizados no ecossistema Internet das Coisas (IdC), estando equipados com sensores e atuadores, e sendo capazes de executar pequenas tarefas com um consumo de energia muito reduzido. Diante de tais dispositivos, a possibilidade de executar uma Rede Neural Profunda (RNP) parece inviável, mas, com as estratégias certas, é possível potenciar as capacidades destes dispositivos e orquestrá-los de modo a realizar esta tarefa. Este trabalho explora o potencial da inferência distribuída, onde um modelo de RNP é particionado e distribuído por múltiplos dispositivos com recursos limitados para alcançar uma inferência de alto desempenho no ambiente de edge. Apresentamos o Distributed Resource Utility & Performance Testbed (DisRUPT), um ambiente de testes de inferência distribuída que permite a avaliação de diferentes estratégias de partição e cenários de implantação para modelos de RNP, incorporando conjuntamente dispositivos far-edge e edge, medindo latência de inferência e ao consumo de energia da rede. Adicionalmente, apresentamos um conjunto de estratégias aplicadas a diferentes arquiteturas de RNP e avaliamos os seus efeitos na latência de inferência e no consumo de energia da rede. Em concreto, conseguimos executar uma adaptação do modelo MobileNetV1 numa rede composta por 2 microcontroladores e 1 Raspberry Pi 3B, executando uma inferência em 760,32 ms (33,78 ms de inferência ativa e 726,54 ms de latência residual), utilizando apenas 145,7 mJ de energia. A partir das nossas configurações com o modelo Depthwise Separable Convolutional Neural Network (DS-CNN) obtivemos resultados que variaram entre uma inferência de baixa latência, demorando 27,99 ms, e uma inferência de baixo consumo energético, utilizando 26,5 mJ de energia por inferência. A partir dos resultados recolhidos neste trabalho, identificamos um conjunto de boas práticas para a inferência distribuída em dispositivos far-edge e edge e, principalmente, apontamos a existência de um limiar de taxa de transferência de entrada onde a inclusão de dispositivos edge no processo de inferência só se torna vantajosa acima deste limiar.

Palavras-chave: Ambiente de Borda • Inferência Distribuída • Internet das Coisas • TinyML

Abstract

In a world where Artificial Intelligence (AI) is increasingly integrated into everyday applications, the strategy for developing and deploying state of the art AI models has been relying heavily on powerful cloud-based servers. However, this approach has several drawbacks, including the distance between the data source, the server and the end-user, which can lead to increased latency, higher energy consumption, and potential privacy concerns. To address these issues, edge computing has emerged as a promising solution, bringing computational resources closer to where they are needed, in particular, the TinyML paradigm, which provides a possibility to run AI models on everyday devices. Nonetheless, far-edge devices, such as microcontrollers, although highly memory and computationally constrained, are widely used in the Internet of Things (IoT) ecosystem, equipped with sensors and actuators, and are capable of running small tasks while consuming very low power. Faced with such devices, the possibility of running a Deep Neural Network (DNN) seems infeasible, but with the right strategies, it is possible to leverage the capabilities of these devices and orchestrate them to achieve this task. This work explores the potential of distributed inference, where a DNN model is partitioned and distributed across multiple resource-constrained devices to achieve highly performant inference in the edge environment. We present Distributed Resource Utility & Performance Testbed (DisRUPT), a distributed inference testbed, that enables the evaluation of different partitioning strategies and deployment scenarios for DNN models incorporating far-edge and edge devices together, in regards of inference latency and network energy consumption. Additionally, we present a set of strategies applied on different DNN architectures, and evaluate their effects on inference latency and the network energy consumption. In particular, we managed to run an adapted MobileNetV1 model on a network of 2 microcontrollers and 1 Raspberry Pi 3B, executing an inference in 760,32 ms (33,78 ms of active inference and 726,54 ms of overhead latency), using only 145,7 mJ of energy. From our setups with the Depthwise Separable Convolutional Neural Network (DS-CNN) model we obtained results that ranged from low latency, taking 27,99 ms to low energy consumption inference, using 26,5 mJ of energy per inference. From the collected results in this work, we identify a set of best practices for distributed inference in far-edge and edge devices, and, mainly, point out the existence of an input throughput threshold where the inclusion of edge devices in the inference process only becomes advantageous above the threshold.

Keywords: Distributed Inference • Far-Edge Environment • Internet of Things • TinyML

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (**SDGs**) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (<https://sdgs.un.org/goals>) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The specific Sustainable Development Goals mentioned have the following names:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.3	This work helps identify which approaches use less energy per inference, contributing to more energy-efficient AI deployments.	Energy consumed (in Joules) per inference, compared across partitioning strategies
9	9.5	A general way to split a neural network across several small, memory-constrained devices. This can be reused and extended in future work on distributed inference.	Number of partitioning heuristics proposed and validated; reduction in per-device memory footprint required to run a given model
	9.c	Lowering of the hardware barrier to deploying AI, making it more accessible to settings with limited budgets.	Total hardware cost of the deployed setup compared to the cost of equivalent cloud-based inference
12	12.2	Reduces dependency on centralized compute resources by distributing AI inference workloads to edge devices for non-essential tasks.	Total energy consumption of the distributed pipeline compared to an estimated equivalent cloud-based inference round trip

Acknowledgements

This work is the crowning achievement of a long journey that I have been on for the last five years. Throughout this time, I have had the opportunity to meet and work with many people to whom I owe a great deal of gratitude.

First and foremost, I would like to thank Professor Filipe Sousa and Fraunhofer Portugal for the opportunity to work on this project and for their availability to help me embark on this project.

Recalling this last year, I would like to thank my supervisors, Fernando Rego, who has been a great guide through this convoluted path where I started without much knowledge about it; and Luís Almeida, who did not hesitate to give me advice whenever I had doubts about the dissertation's specifics. As I often joked this year, "this is my first dissertation".

Although I have had a lot on my plate, I have always had the support of my friends, who were my family away from home. I would like to thank them for their aid in lightening the mood and helping me get through the worries inherent to this work, always giving me an extra hand when I needed it.

Central to this journey has been the unwavering support of my family, who have always been there for me, even when I was not sure about what I wanted to do with my career. I am grateful for their unconditional support and love, which was above all a great source of motivation for me to keep going and finish this work.

Tomás Vicente

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use was limited to writing revision and enhancement, starting from a draft of my own authorship, and its results have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments.

Tomás Alexandre Soeiro Vicente



*“Para fazer qualquer coisa é preciso primeiro a audácia de a começar,
e depois a paciência de a terminar.”*

Ditado popular

Contents

1	Introduction	1
1.1	Problem	1
1.2	Objective and Research Questions	2
1.3	Dissertation Structure	2
2	Literature Review	4
2.1	Introduction	4
2.2	Review Methodology	4
2.2.1	Search Strategy	5
2.2.2	Inclusion and Exclusion Criteria	5
2.3	Search Results	6
3	State Of The Art	8
3.1	System Network Layouts	8
3.1.1	Hierarchical Inference	8
3.1.2	Model Chain Partitioning	9
3.1.3	Model Local Partitioning	11
3.2	Network Node Management	12
3.2.1	Heuristical Approach	12
3.2.2	Reinforcement Learning Approach	14
3.3	Final Remarks	16
4	Distributed Resource Utility & Performance Testbed	20
4.1	System Overview	20
4.2	Model Partitioning	21
4.3	MCU Interface	22
4.4	Raspberry Pi Interface	22
4.5	Deployment Planning	23
4.5.1	Device Communication	23
4.5.2	Performance Metrics	26
4.5.3	Partition and Deployment Strategies	26
5	Experiments and Results	29
5.1	Experimental Setup	29
5.1.1	Network Topology	29
5.1.2	Energy Measurement	30
5.2	Models	31
5.2.1	All-CNN-C	31

5.2.2	DS-CNN	35
5.2.3	MobileNetV1	39
5.2.4	MLP Autoencoder	43
5.3	Overall Results Analysis	47
6	Conclusion	49
6.1	Contributions	49
6.2	Future Work	50
	References	51
A	Model Specifications	54
B	Model Energy	59
B.1	Submodel Energy Consumption Overview	59

List of Figures

2.1	Screening process of the systematic literature review.	6
4.1	Distributed Resource Utility & Performance Testbed (DisRUPT) architecture. . .	21
4.2	Raspberry Pi Interface architecture.	23
4.3	Communication protocol landscape for Internet of Things (IoT) (from Gulati et al. [9]).	25
5.1	All-CNN-C architecture diagram from Buda [5].	32
5.2	All-CNN-C submodel memory footprints.	32
5.3	All-CNN-C inference latency measurements.	34
5.4	All-CNN-C energy consumption measurements.	35
5.5	DS-CNN architecture diagram from Sørensen, Epp, and May [22].	36
5.6	DS-CNN submodel memory footprints.	37
5.7	DS-CNN inference latency measurements.	38
5.8	DS-CNN energy consumption measurements.	39
5.9	MobileNetV1 depthwise separable convolution structure from Nguyen [16]. . . .	40
5.10	MobileNetV1 submodel memory footprints.	41
5.11	MobileNetV1 inference latency measurements.	42
5.12	MobileNetV1 energy consumption measurements.	42
5.13	Autoencoder MLP architecture from Halder, Fidkowski, and Maki [10].	43
5.14	MLP Autoencoder submodel memory footprints.	44
5.15	MLP Autoencoder inference latency measurements.	46
5.16	MLP Autoencoder energy consumption measurements.	47
A.1	All-CNN-C layer overview.	54
A.2	DS-CNN layer overview.	55
A.3	MobileNetV1 layer overview (part 1).	56
A.4	MobileNetV1 layer overview (part 2).	57
A.5	MLP-Autoencoder layer overview.	58

List of Tables

2.1	Inclusion and exclusion criteria for the systematic literature review.	5
2.2	Number of documents found per topic and digital library.	6
3.1	Overview of the system layouts of the reviewed strategies.	13
3.2	Overview of the network management in the reviewed strategies.	16
3.3	Overview of the reviewed strategies.	17
4.1	Raspberry Pi Interface components.	24
4.2	Deployment strategies for model partitioning and segment assignment.	28
5.1	All-CNN-C distribution configuration.	33
5.2	DS-CNN distribution configuration.	37
5.3	MobileNetV1 distribution configuration.	41
5.4	MLP Autoencoder distribution configuration.	45
B.1	Energy consumption overview by model, partition, and device	60

List of Acronyms

AI	Artificial Intelligence
CBOR	Concise Binary Object Representation
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
DisRUPT	Distributed Resource Utility & Performance Testbed
DL	Deep Learning
DRL	Deep Reinforcement Learning
DNN	Deep Neural Network
DS-CNN	Depthwise Separable Convolutional Neural Network
FC	Fully Connected
FLOP	Floating Point Operation
GPU	Graphics Processing Unit
IoT	Internet of Things
JSON	JavaScript Object Notation
MCU	Microcontroller Unit
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilayer Perceptron
MPU	Microprocessor Unit
NAMP	Network-Aware Model Partitioning
NPU	Neural Processing Unit
PPO	Proximal Policy Optimization
QoS	Quality of Service
RPC	Remote Procedure Call
SDG	Sustainable Development Goal
TinyMLaaS	TinyML-as-a-Service
TPU	Tensor Processing Unit
IA	Inteligência Artificial
IdC	Internet das Coisas
RNP	Rede Neural Profunda

Chapter 1

Introduction

For the last couple of years, the field of Machine Learning (ML), and, in particular, its Deep Learning (DL) branch has been a constant presence in the computational research breakthroughs, each time creating new mechanisms to detect more patterns and allowing the extraction of knowledge from seemingly insignificant data, introducing new ways of understanding the world around us [20].

In a similar fashion, the Internet of Things (IoT) paradigm has been gaining more and more traction, making use of numerous interconnected devices that gather extensive data from the environment and provide a wide range of services to users. We can find IoT devices in our homes, workplaces, and cities, where they continuously monitor and process information to perform their designated tasks.

Deep Neural Network (DNN) models can be used in a wide range of applications, from automotive and healthcare to smart cities and industry 4.0 [11]. By combining these two paradigms, we can extract even more benefits, as ML models deployed on IoT devices enable faster, more secure and more accurate data processing and analysis [1].

These IoT devices are often referred to as *edge* or *far-edge devices*. Still, currently, there isn't a clear and agreed upon definition of what edge and far-edge devices are. Nonetheless, for this work, a far-edge device can be seen as a constrained device, characterized by limited computational power and memory, such as sensors or other Microcontroller Unit (MCU) based devices. An edge device, similarly, has some constraints, but is more powerful than far-edge devices, these devices can be, for example, a Microprocessor Unit (MPU) based device. The latter often serves as the connection point between far-edge devices and the cloud.

1.1 Problem

To provide these tools to users, the most common approach is to use cloud computing infrastructures, where the data is sent to a remote server for processing and the results are sent back to the user. However, this approach presents some limitations, such as latency, privacy and security concerns, and the need for a constant internet connection. Furthermore, the increasing amount of

data generated by **IoT** devices can inflict a significant burden on the cloud infrastructures, as the services find themselves struggling with bandwidth and scalability issues.

To overcome these limitations, the concept of edge computing has been introduced, where the data is processed closer to the source, on edge devices, such as **MCUs**. However, these devices have limited computational resources, which can make it challenging to deploy complex **DNN** models on them [24].

Moreover, there is a lack of widely spread and standardized tools and frameworks to facilitate the usage of **ML** in constrained devices, as they are constituted by a heterogeneous set of hardware architectures and software environments. Works like TinyML-as-a-Service (**TinyMLaaS**) [11] try to address this, but they still face limitations in their applicability and ease of use.

Some other efforts were made to address this problem, such as NAMP [8], which proposes a method for partitioning **DNN** models between **MCUs**. However, these advancements only scratch the surface of the problem and still require improvement in their scalability potential and communication protocols.

1.2 Objective and Research Questions

Considering the context and the problem presented, this work aims to answer the following hypothesis:

“Is it possible to create a reliable tool to support the deployment of complex **DNN** models on far-edge and edge devices?”

As the hypothesis tackles a complex problem, there are several research questions that can be derived from it as intermediate steps to answer it, further completing the understanding of the problem and guiding the research process.

RQ1: Is it feasible to have collaborative **ML** inferences using edge and far-edge devices¹ without compromising performance?

RQ2: What is the most efficient method to enable reliable cooperation between edge and far-edge devices partaking in an inference network?

RQ3: How efficiently can the resources of edge and far-edge devices be utilized towards collaborative inference tasks?

1.3 Dissertation Structure

In addition to the introduction, this dissertation contains five more chapters.

¹Let it be known that the expression *inference networks* refers to the collaborative execution of a **DNN** model between multiple devices.

In Chapter 2, the literature review procedure is described and the outcome analyzed. While in Chapter 3 the state of the art is described, and related work is presented and categorized. Chapter 4 presents the developed approach that is tested in Chapter 5. This work is then concluded with Chapter 6, where the main contributions are summarized, and future work is proposed.

Chapter 2

Literature Review

This chapter investigates existing research on deploying **DNN** models on edge and far-edge devices through a systematic literature review. We examine three main areas: model partitioning and split inference, **ML** orchestration and scheduling in edge computing, and computational offloading for **ML** inference. The chapter details the search methodology, including the search strategy and inclusion/exclusion criteria, and presents the results, establishing the current state of the art in these fields.

2.1 Introduction

This literature review followed Kitchenham et al. [14]’s Systematic Literature Review workflow. The review aimed to identify and analyze existing research relevant to the subject of this thesis. Likewise, it was designed to provide a strong foundation for the following topics:

1. Model Partition and Split Inference
2. **ML** Orchestration/Scheduling in Edge Computing
3. Computational Offloading for **ML** inference

In addition, the proposal bibliography was also included and examined in the context of this work.

2.2 Review Methodology

To ensure replicability and rigor, the following subsections describe the methodology employed to retrieve relevant studies. Particular emphasis is placed on the search strategy, including the search queries used, and the criteria applied for filtering results.

2.2.1 Search Strategy

There were three digital libraries used to search for relevant articles: the IEEE Xplore Digital Library¹ and the ACM Digital Library² as the main sources of specialized literature on the subject, and Scopus³ as a broader database to cover any gaps that could arise.

To accomplish this search, the following search strings were built, respectively for each topic:⁴

1. ("dynamic partitioning" OR "dynamic splitting" OR "model partitioning" OR "model splitting" OR "adaptive partitioning" OR "adaptive splitting") AND "inference" AND ("ML model*" OR "DNN") AND ("Edge" OR "Far-Edge")
2. (("resource-aware" OR "network-aware" OR "adaptive") NEAR/5 ("schedul*" OR "orchestrat*" OR "allocation")) AND ("inference" OR "ML model*" OR "DNN" OR "AI workload*") AND ("Edge" OR "IoT" OR "Far-Edge")
3. ("inference offload*" OR "model offload*" OR "task offload*") AND ("resource-aware" OR "latency-aware" OR "adaptive" OR "resource-constrained") AND ("TinyML" OR "MCU" OR "microcontroller*")

2.2.2 Inclusion and Exclusion Criteria

In order to filter the search results, the rules in Table 2.1 were applied to include or exclude the documents found:

Type	ID	Criteria
Inclusion	IC1	The publication is in an adequate document type.
	IC2	Date of publication since 2018 (inclusive).
	IC3	The publication addresses constrained devices.
	IC4	The publication discusses network deployment or topics pertinent to it.
Exclusion	EC1	It is a duplicate result.
	EC2	Full-text is not available.
	EC3	The publication is not English.
	EC4	The publication focuses on cloud computing.
	EC5	The publication focuses on high-performance device usage.

Table 2.1: Inclusion and exclusion criteria for the systematic literature review.

¹<https://ieeexplore.ieee.org/Xplore/home.jsp>

²<https://dl.acm.org/>

³<https://www.scopus.com/>

⁴It should be noted that the proximity and wildcard operators were adapted to each digital library syntax.

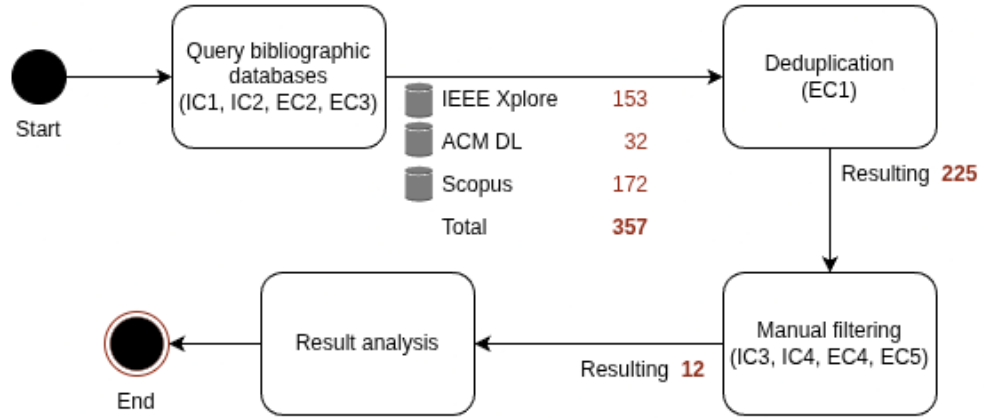


Figure 2.1: Screening process of the systematic literature review.

Different formats were found in the indicated set of digital libraries. The following formats were considered adequate for this review: *Conference Paper*, *Journal* and *Magazine* for *IEEE Xplore*; *Research-Article*, *Demonstration* and *Short-Paper*, for *ACM Digital Library*; *Article*, *Conference Paper* and *Conference Review*, *Book Chapter*, *Review* and *Book* for *Scopus*. The document type filter also excluded *Work-in-Progress* documents.

2.3 Search Results

The literature search was fruitful, returning a total of 357 documents across the three topics and the three digital libraries used.

Table 2.2 summarizes the number of documents found on the initial stage per topic and digital library, yet to be filtered through the inclusion and exclusion criteria.

Topic	IEEE Xplore	ACM DL	Scopus	Total
1	54	24	71	149
2	93	8	95	196
3	6	0	6	12
Total	153	32	172	357

Table 2.2: Number of documents found per topic and digital library.

After applying the inclusion and exclusion criteria, a total of 12 documents were included for further analysis and serve as basis for this work. Figure 2.1 illustrates the flow that documents went through during the screening process, as listed below.

- Al-Atat et al., “*The Case for Hierarchical Deep Learning Inference at the Network Edge*” (2023)
- Behera et al., “*Improved Decision Module Selection for Hierarchical Inference in Resource-Constrained Edge Devices*” (2023)

- Chen et al., “*Deep Reinforcement Learning for Collaborative Inference in Local- Edge Computing*” (2024)
- E. M. K et al., “*Adaptive Hierarchical Inference Framework for Enhanced Efficiency and Scalability in Complex IoT Decision-Making Systems*” (2025)
- F. Rego and Almeida, “*NAMP: A Network-Aware Model Partitioning Framework for Constrained Devices*” (2025)
- Ma, Wang, and Tang, “*Joint Optimization of Model Partitioning and Resource Allocation for Multi-Exit DNNs in Edge-Device Collaboration*” (2025)
- P. Dai et al., “*Joint Optimization of Device Placement and Model Partitioning for Cooperative DNN Inference in Heterogeneous Edge Computing*” (2025)
- R. G. Pacheco, R. S. Couto, and O. Simeone, “*Calibration-Aided Edge Inference Offloading via Adaptive Model Partitioning of Deep Neural Networks*” (2021)
- T. Mohammed et al., “*Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading*” (2020)
- Tran-Van and Le, “*CoFANN: A collaborative framework for accelerating DNN inference in drone-based agricultural monitoring systems*” (2025)
- W. Miao et al., “*Adaptive DNN Partition in Edge Computing Environments*” (2020)
- Wang et al., “*Context-aware Adaptive Surgery: A Fast and Effective Framework for Adaptive Model Partition*” (2021)

Chapter 3

State Of The Art

In this chapter, we analyze the current state of the art described by the documents gathered in the literature review (Chapter 2). We will review different approaches and techniques applied in Edge and IoT environments to achieve a better performance when executing ML inference tasks. In particular, the overall layout and flow of the systems, recognizing the role of each element that takes part in the inference task, as well as the management and orchestration of the network, analyzing the strategies for scheduling and communication methods.

3.1 System Network Layouts

Firstly, we will analyze how the proposed frameworks make use of different system layouts to achieve collaborative inference in edge and far-edge devices. We will classify them into three main categories, according to their system layout: Hierarchical Inference, Model Chain Partitioning and Model Local Partitioning.

3.1.1 Hierarchical Inference

Rather than focusing on partitioning a single monolithic model, these frameworks deploy differently sized ML models throughout several hierarchies of devices according to their computational capabilities (i.e., deploying smaller models closer to the edge, and larger and more capable ones closer to the cloud).

Hierarchical inference is a strategy that relies on the deployment of multiple models with different sizes and complexities across a hierarchy of devices. Its most basic form involves deploying a smaller, less complex model on the edge device, to perform an initial inference. The results of this initial inference are then analyzed by a decision module, which determines whether the edge device is confident enough to classify the input data. If the confidence threshold is not met, the data is transmitted to an edge server, where a large DL model is deployed to perform a more accurate inference.

Al-Atat et al. [2] present an implementation of the idealized hierarchical inference system, such that it works as an image classification system in a real-world scenario. The smaller model is made of five layers and built using TensorFlow and later quantized to TFLite. On the other hand, for the larger model, EfficientNet was chosen, achieving a 95% accuracy. Several methods for offloading decisions were tested, including a simple confidence threshold and a more complex minimization of a cost function that takes into account the offload and misclassification costs. The framework achieves a significant reduction in latency, energy consumption and bandwidth usage, at the cost of a small reduction in accuracy.

Behera et al. [3] reinforce all the ground knowledge for Hierarchical Inference, by providing a comprehensive review of the intended purpose of the strategy, such as an initial layer that is more efficient in the inference task than plainly offloading it to an Edge Server, in terms of energy consumption, latency, and memory constraints. Additionally, the work further explores methods to improve the decision module by testing different methods, such as a model calibration with fixed threshold, to face the overconfidence problem; use of classifiers after the TinyML model, that take the final layer of the TinyML model to make a more informed decision; and the use of classifiers before the TinyML model, that tries to predict if the sample is hard to classify, and therefore should be directly offloaded to the Edge Server, avoiding the energy consumption of the initial inference. In the end, the work concludes that the use of a Linear Regression model after the TinyML model is the best approach, considering the overall costs and the evaluation metrics used.

E. M. K et al. [7] propose a more complex hierarchical inference system, including three layers of models, deployed on the end device, edge server and cloud. The framework consists of an end layer composed of single-board computers (e.g.: ESP32, STM32), an edge layer composed of more powerful devices (e.g.: Raspberry Pi, Jetson Nano) and a cloud layer where models like Gemini and ChatGPT are hosted. This framework is designed to be highly adaptable, allowing for a quick response to simple inputs, while also being able to handle more complex inputs by offloading them to more powerful devices. Although the framework exhibits a more robust structure, the offloading decision module is as simple as a confidence threshold, which may lead to suboptimal performance.

3.1.2 Model Chain Partitioning

The approaches reviewed in this category divide the workload through a chain of devices, deploying the initial portion of the inference task to devices closer to the end-user and the remaining portion to one or more devices on a higher hierarchical layer, further away from the end-user device. Assuming a device that is further away from the end-user device has more computational capabilities.

CoFANN [26] addresses the trade-off between model accuracy and efficiency by employing dynamic inference strategies. CoFANN significantly reduces overall latency through search

algorithms that minimize additional latency overhead. The inference workload is divided between **IoT** devices and edge devices. The network devices are benchmarked in an initial phase, allowing the framework to estimate the time needed for each device on each step of the model. The framework then employs a differentiable search algorithm to find optimal inference strategies for allocating **DNN** tasks to the computing devices. Even though this framework deploys successfully a partitioned model between an end device and an edge server, it is limited to two-device networks.

DINA [25] is a framework that relies on fog nodes to partition inference tasks into sub-tasks and deploys them according to each node networking conditions, using matching theory for joint partitioning and task offloading. This framework allows for a more efficient use of the network resources, while also reducing the total computation time. This work can process **DNN** models with a Directed Acyclic Graph (**DAG**) topology, composed of fully connected and convolutional layers. The framework determines how to partition the inference task and then divides it according to the network condition. It should be stated that the network is expected to be homogeneous by design, as it does not take the device characteristics into account. Furthermore, the performance evaluation was conducted through simulations, which may not fully capture real-world scenarios.

Chen et al. [6] present a framework that maximizes the user Quality of Service (**QoS**), compared to traditional frameworks, as it allows for the user to specify their priorities, such as inference latency and energy consumption. This is achieved by modelling the collaborative inference problem as a Markov Decision Process (**MDP**) and applying a Deep Reinforcement Learning (**DRL**)-based algorithm to determine the optimal partitioning point (i.e., the last subtask in a given **DNN** model), between end and edge devices. This way the user requirements and the network conditions, such as device capabilities and network bandwidth, are taken into account to optimize the inference process.

Ma, Wang, and Tang [15] propose a framework that focuses on minimizing energy consumption, making use of multiple early-exits along the inference process, when confidence thresholds are met. The framework partitions each model instance into two segments, depending on the end device capabilities. The initial segment is deployed on the end device, while the remaining segment is deployed on an edge server. The framework deploys the same model architecture across a range of devices participating in the inference task, adapting each instance to the device capabilities. The early-exit mechanism allows for task termination prematurely in early stages of the inference task, reducing energy consumption and latency, given the model is confident enough.

R. G. Pacheco, R. S. Couto, and O. Simeone [19] develop a framework that exploits the early-exit mechanism to reduce the inference delay in the inference process. This is achieved by adding a classifier branch at an intermediate layer of the **DNN** model, allowing the model to make a prediction with a given confidence at that point. If the confidence threshold is met,

the inference task is terminated early, otherwise, the remaining layers are processed in the cloud, after the intermediate results are transmitted. This framework presents a clear implementation of the early-exit mechanism, surpassing the related overconfidence problem in the intermediate classifier by applying a calibration technique, such as temperature scaling. Nonetheless, the framework lacks scalability, as it is only applied to a two-device network.

3.1.3 Model Local Partitioning

In this subsection, we will analyze approaches that deploy the whole partitioned system to devices locally, without requiring computing power from nodes closer to the cloud.

CAS [28] is a framework that successfully adapts a given model in real-time, upon a network context change. It generates a "partition state graph", containing the set of possible partitions, given a **DNN** model, choosing one based on the current network status, such as inference latency and resources budgets. This work gathers the information of the network, taking note on possibly constrained devices. This framework creates a fast adapting system, responding to network fluctuations, keeping the active deployment optimized.

NAMP [8] is a framework that emphasizes resource utilization, by partitioning the model into several segments and deploying them to multiple severely constrained devices. The framework analyzes the network devices and their capabilities, partitioning the model into appropriate sized segments and deploying them to the devices, according to their capabilities. The framework allows for inference tasks to be executed in a highly scalable manner, even in an environment of highly constrained devices, although it is limited in the model topology flexibility, as it only supports chain-like **DNN** models.

P. Dai et al. [18] present a solution to the collaborative inference problem that optimizes device placement and model partitioning, accounting for particularities of heterogeneous edge resources and Convolutional Neural Network (**CNN**) models, with the objective to maximize throughput. A combination of intra-layer and interlayer partitioning techniques are applied in the proposed approach, including layer-wise partitioning for interlayer partitioning, and channel and spatial segmentations for intra-layer partitioning. This approach reformulates the network as a pipeline system, where each device is responsible for simultaneous inference batches. The **DNN** model segments are tailored for the devices in the network and assigned according to the network topology, that can be represented as a **DAG**. The devices in the network can have sequential or parallel relationships, that affect the assigned model segment.

W. Miao et al. [27] explore an approach based on splitting and deploying a given **DNN** model with a **DAG** topology to several devices, exploiting parallelizable segments (branches), based on model data and computation time estimations for each layer and branch processing. The characteristics of this network are also taken into account, such as, its bandwidth

and the number of devices present in the network. The authors also allow the network to be based on devices with homogeneous computation times or heterogeneous, given the computation time estimations of each layer and branch for each device. This work also points out the unreliability and security concerns in partitioning the model between data centers and edge devices.

Unlike traditional cloud-based inference systems, the reviewed frameworks leverage the computational capabilities of edge and far-edge devices to distribute the inference workload. This distribution not only reduces latency but also optimizes resource utilization across the network, allowing for a faster and more power efficient processing.

Several techniques can be employed to achieve this effect, such as model partitioning, which can be done in different ways: either by bi-partitioning the model between far-edge and edge devices, or by locally partitioning the model among a collection of devices closer to the far-edge layer. On the other hand, some frameworks employ partitioning on a per-layer basis, while others rely on graph-based partitioning techniques to better adapt to the model topology.

Another important architecture used to achieve a more efficient inference process is Hierarchical Inference, which is accomplished by deploying differently sized ML models through several hierarchies of devices, allowing for a quick response in simple inputs where a smaller model is sufficient.

Characteristics of the latter strategy can be found in the partitioning based approaches, materialized in the early-exit mechanisms. Early-exits allow the load to be distributed, reaching only the more powerful devices when the inference task is more complex, while still reducing the overall latency, energy consumption and bandwidth usage.

To better standardize the following synthesis, in the Table 3.1, we will recognize previously referred user, end devices and similarly constrained devices as our definition of far-edge devices. Likewise, the intermediate devices, placed higher in the hierarchy than the far-edge devices (e.g.: edge nodes, local servers), will be recognized simply as edge devices.

3.2 Network Node Management

In a distributed system, the orchestration and management of every node of a given network is of utmost importance. Since we are reviewing collaborative frameworks, we will be analyzing how the presented frameworks strategize the scheduling and communications between their devices. Furthermore, the frameworks will be separated into two classes, heuristical and reinforcement learning scheduling approaches.

3.2.1 Heuristical Approach

Predicting the assignments that lead to the most efficient processing of any given inference task has been revealed to be a highly complex problem. Therefore, several works have relied on heuristical approaches to simplify the problem avoiding the loss of a big part of the desired efficiency. In

Strategy	System Layout	System Agents	Partitioning Method	Uses Early Exits
Al-Atat et al. [2]	Hierarchical Inference	Far-Edge Devices, Edge Devices	N/A	✓
Behera et al. [3]	Hierarchical Inference	Far-Edge Devices, Edge Devices	N/A	✓
CAS [28]	Model Local Partitioning	Edge Devices	Graph Partitioning	✗
Chen et al. [6]	Model Chain Partitioning	Edge Devices	Graph Partitioning	✗
CoFANN [26]	Model Chain Partitioning	Far-Edge Devices, Edge Devices	Graph Partitioning	✗
DINA [25]	Model Chain Partitioning	Far-Edge Devices, Edge Devices	Graph Partitioning	✗
E. M. K et al. [7]	Hierarchical Inference	Far-Edge Devices, Edge Devices, Cloud	N/A	✓
Ma, Wang, and Tang [15]	Model Chain Partitioning	Far-Edge Devices, Edge Devices	Inter-Layer Partitioning	✓
NAMP [8]	Model Local Partitioning	Far-Edge Devices	Inter and Intra-Layer Partitioning	✗
P. Dai et al. [18]	Model Local Partitioning	Edge Devices	Inter and Intra-Layer Partitioning	✗
R. G. Pacheco, R. S. Couto, and O. Simeone [19]	Model Chain Partitioning	Edge Devices, Cloud	Inter-Layer Partitioning	✓
W. Miao et al. [27]	Model Local Partitioning	Far-Edge Devices	Graph Partitioning	✗

Table 3.1: Overview of the system layouts of the reviewed strategies.

this subsection we will be reviewing the frameworks that rely on heuristics, which heuristics are chosen and how the network is managed.

CAS [28] makes use of "the neighbor effect" and assigns the initial partitions to the first devices, closer to the user, and allocates the tail layers to devices closer to the cloud, noted to be heavier on computation time even if lighter on the data. All the devices' constraints are gathered in a previous stage and taken into account. Network conditions are monitored in real-time, allowing the framework to strive for an optimal partitioning and deployment, adapting to network changes, such as device power and network bandwidth fluctuations, as well as user requirement of latency.

CoFANN [26] is a minimalist framework, in regard to network management, taking only into

account a pair of devices. Nonetheless, it looks to minimize inference time and data transfers by searching thoroughly a continuous search space created from factors in the inference process. To minimize the data transfer, the initial raw data is processed in the end device and condensed as an intermediate feature tensor to be wirelessly transmitted to an edge server.

DINA [25] approaches the scheduling problem using matching games, based on game theory. Matching is obtained according to preference relations, similarly to partitioning, which are derived by means of utility functions, representing the benefit obtained by executing a given task on that device. The initial matching is further improved by proposing swapping edge nodes between two user nodes, if it benefits both parties.

NAMP [8] implements a heuristical scheduling algorithm to optimize the inference process, taking into account the chosen programmable heuristics, such as inference time or resource utilization. The framework gathers the devices' characteristics and determines the optimal partitioning and deployment according to the chosen heuristics only before deployment, without further responsiveness to network changes. The deployment is achieved by using the Zephyr RTOS¹ with the CoAP protocol, to further minimize the overhead of the communication between devices.

P. Dai et al. [18] use device placement to assign the model partitions to the devices. The feature extractor segment of the **CNN** models is distributed along the network, while the classifier, as it requires less computation, is assigned to a singular device, given it has sufficient available memory for the corresponding weights. Intermediate inference results are exchanged between devices through a wireless link, allowing for a higher throughput. This solution assumes that the pipeline supports asynchronous communication.

W. Miao et al. [27] minimize the inference time relying on estimations of the computing time needed at each step. Since the inference time is determined by the edge device that gets results last, they approach scheduling by reducing it to a load balancing problem. The data offloading between devices is implemented by Remote Procedure Call (**RPC**)s.

3.2.2 Reinforcement Learning Approach

Rather than relying on simple heuristics to achieve a near-optimal solution, the works in this subsection build reinforcement learning models to capture the complex relationship between the network and the optimal assignment to a greater extent.

Chen et al. [6] employ a **DRL**-based algorithm to determine the optimal partitioning point, between end (our "Far-Edge") and edge devices. The work elaborates a special algorithm that branches from the traditional Deep Q-Network (DQN), to improve decision-making efficiency, by storing and reusing past experiences, as well as by having a dual estimation stream, one for state-value function and one for action-value function. Nevertheless, the

¹Linux Foundation, Zephyr Project, <https://www.zephyrproject.org/>

framework also regards the user priorities, integrated in experience memory sampling. This allows the framework to minimize the inference latency or the energy consumption, according to the user needs. The network also makes use of ZeroMQ² for communication between devices, using a PUSH-PULL pattern for data transfer.

Ma, Wang, and Tang [15] make use of a **DRL**-based algorithm, employing an actor-critic model to determine the optimal resource assignment and the layer of the inference task to mark the bi-partitioning point. The Proximal Policy Optimization (**PPO**) algorithm is used to train the model, optimizing cost evolution, balancing energy consumption and inference latency. This approach takes into account all relevant system information, including task characteristics, device capabilities, and network conditions. These conditions include the layer required Floating Point Operations (**FLOPs**), computational capabilities, power consumption at each device, network channel bandwidth, among other relevant factors. This information is used to make informed decisions on resource allocation and task partitioning.

To manage efficiently a network of devices, it is crucial to have a robust strategy for scheduling and communication. The reviewed frameworks employ different approaches to tackle this problem, ranging from simpler heuristics to more complex reinforcement learning algorithms.

On one hand, heuristical approaches provide a straightforward and computationally light solution to the scheduling problem, while also being deterministic. They often rely on predefined rules or optimization techniques to allocate tasks and manage communication between devices, avoiding the need for more complex computation as required by the reinforcement learning approaches (i.e., training). These methods thrive on scenarios where the network conditions are relatively stable and predictable. However, they may struggle to adapt to dynamic environments or complex network topologies, lacking the overall flexibility and robustness.

The most common approaches to these heuristic scheduling algorithms are based on estimating the computational load and communication overhead for each device and subtask, and then using a search algorithm to find a near-optimal assignment of tasks to devices. This method, if not adapted, may lead to suboptimal performance, as the estimations end up overlooking any particularities of the devices and the network, including fluctuations in performance, transmission and other factors that may further hinder the system performance.

On the other hand, reinforcement learning approaches offer a more sophisticated solution by allowing the system to learn optimal scheduling policies through interaction with the environment. These methods can adapt to changing network conditions and can potentially discover more efficient scheduling strategies. However, they often require more computational resources and time to train the learning models, as well as enough data from the network to make informed decisions.

The communication between devices is another critical aspect of network management. Efficient communication protocols and methods can significantly impact the overall performance of the collaborative inference process. The reviewed frameworks utilize various communication methods, even if not always explicitly specified (i.e., the wireless links referred to in Table 3.2).

²<https://zeromq.org/>

These methods include ZeroMQ, CoAP protocol, and **RPCs**, each with the goal of minimizing communication overhead and latency while ensuring reliable data exchange.

The previously described frameworks that were not mentioned in either of these network management categories do not present any particular strategy for network management, relying on previous works or simple approaches.

Strategy	Scheduling Approach	Optimized Metrics	Communication Method	Tolerates Node Heterogeneity
CAS [28]	Heuristical	Inference Time	Wireless link (3G, 4G)	✓
Chen et al. [6]	Reinforcement Learning	User QoS	ZeroMQ PUSH-PULL	✓
CoFANN [26]	Heuristical	Inference Time, Data Transferred	Wireless link	✓
DINA [25]	Heuristical	Inference Time, Resource Utilization	Wireless link	✗
Ma, Wang, and Tang [15]	Reinforcement Learning	Inference Time, Energy Consumption	N/A	✓
NAMP [8]	Heuristical	Configurable Heuristic	CoAP protocol	✓
P. Dai et al. [18]	Heuristical	Device Placement	Wireless link	✗
W. Miao et al. [27]	Heuristical	Inference Time	RPC	✓

Table 3.2: Overview of the network management in the reviewed strategies.

3.3 Final Remarks

In this chapter, we have analyzed the current state of the art in collaborative inference frameworks for edge and far-edge devices. We have reviewed various approaches and techniques employed to optimize the inference process, focusing on system layouts and network management strategies.

Each framework presents unique solutions to the challenges posed by distributed inference tasks. Some works focus on model partitioning strategies to maximize the use of available resources, while others emphasize efficient scheduling and communication methods to minimize latency and overhead.

We have reviewed frameworks according to their system layouts. Some frameworks rely on Hierarchical Inference, deploying differently sized **ML** models across several sets of devices according to their computational capabilities. Others rely on model partitioning techniques to distribute the inference workload across the network, such as the Model Chain Partitioning and Model

Local Partitioning approaches. The former involves dividing the inference task into segments deployed to different network layers with different computational capabilities. The latter deploys the entire partitioned model locally to a cluster of devices closer to the far-edge layer.

Furthermore, we have analyzed the network management strategies employed by the reviewed frameworks. We categorized them into heuristical and reinforcement learning approaches, providing a range of possible solutions from simple and fast heuristics to more complex and adaptive reinforcement learning algorithms.

The diversity of approaches highlights the complexity of the problem and the need for tailored solutions depending on the specific requirements and constraints of the target environment. An overview of the reviewed strategies is presented in Table 3.3, summarizing their key features and relevant aspects.

Table 3.3: Overview of the reviewed strategies.

Strategy	Overview	Relevant Aspects
Al-Atat et al. [2]	Hierarchical inference system for image classification. Achieves significant reductions in latency, energy consumption and bandwidth usage, at the cost of a small reduction in accuracy.	• Use of a cost function to make off-loading decisions. • Deployment of a smaller model on the edge device and a larger model on the edge server.
Behera et al. [3]	Exploration of Hierarchical Inference strategy implementations, focuses on improving the decision module through different methods. Concludes that the use of a Linear Regression model after the TinyML model is the best approach.	• Overall comparison of different methods to improve the decision module.
CAS [28]	Edge system responsive to network changes, allows for local collaborative inference with low latency withstanding network fluctuations. Deployed on edge devices, avoiding reliance on large computing resources.	• Deployed partition responsiveness to network changes in real-time.
Continued on next page		

Strategy	Overview	Relevant Aspects
Chen et al. [6]	Framework that maximizes user QoS by allowing the user to specify their priorities, such as inference latency and energy consumption. This is achieved by modelling the collaborative inference problem as a MDP and applying a DRL-based algorithm to determine the optimal partitioning point, between end and edge devices.	• User priority integration in experience memory sampling. • Use of ZeroMQ for communication between devices.
CoFANN [26]	Proposes a bi-partitioned system that uses end devices in the network to compute initial inference. The framework relies on only two devices, one on-site and one edge server.	• Employment of a gradient descent based search algorithm to detect the optimal inference strategy, minimizing latency and data transfer.
DINA [25]	Employs a bi-hierarchical fog computing architecture to execute inference tasks. Deploys partitioned DNN models with DAG topologies to homogeneous devices. Adapts to reduce execution time in varying network conditions.	• Extension of a local inference task to a distributed edge device network. • Partition and compression techniques for CNN models. • Matching theory for scheduling purposes.
E. M. K et al. [7]	Hierarchical Inference system with three layers of models, from MCUs to cloud-based models. Highly adaptable framework, although it uses a simplistic offloading decision module.	• Coordination between three layers of models, from MCUs to cloud-based models.
Ma, Wang, and Tang [15]	Framework that minimizes energy consumption in inference networks by employing early-exit mechanisms and bi-partitioning of DNN models. The framework relies on a DRL-based algorithm to optimize resource assignment and partitioning point.	• Use of early-exit mechanisms to reduce energy consumption and latency. • DRL-based scheduling approach using PPO algorithm.
Continued on next page		

Strategy	Overview	Relevant Aspects
NAMP [8]	Framework that emphasizes resource utilization, partitioning the model into several segments, fit for multiple severely constrained devices. The framework is flexible and makes the most use of the network heterogeneous resources, as well as allows for the user to choose the heuristics to optimize.	• Configurable heuristics to optimize. • Use of Zephyr RTOS and CoAP protocol to minimize communication overhead.
P. Dai et al. [18]	Widely applicable framework for collaborative inference in edge networks, optimizing throughput and adapting partitioning according to each device's capabilities. The framework has been shown to be a highly scalable, robust and efficient solution.	• Partition and parallelization techniques for CNN models. • Device placement and network topology considerations for scheduling.
R. G. Pacheco, R. S. Couto, and O. Simeone [19]	Specialized work in early-exit mechanisms for split inference, even if limited to two-device networks. The framework excels in inserting early-exit branches in intermediate layers of DNN models, while addressing the overconfidence problem.	• Effective early-exits insertion mechanism. • Calibration techniques to address overconfidence.
W. Miao et al. [27]	Collaborative inference framework that takes advantage of parallelizable segments in DNN models with DAG topologies. The framework adapts to heterogeneous devices, balancing the load between them to minimize the overall inference time.	• Load balancing approach for heterogeneous devices.

Chapter 4

Distributed Resource Utility & Performance Testbed

To answer the research questions outlined in Chapter 1, we developed the Distributed Resource Utility & Performance Testbed (**DisRUPT**) that can evaluate the performance of distributed resource management strategies. In this chapter, we will discuss the design and implementation of **DisRUPT** using Raspberry Pi¹ devices along with a set of **MCUs**. The Raspberry Pi devices serve as our reference for edge devices, as they are more computationally capable than the **MCUs**, while keeping a low power consumption, opposed to laptop computers, that could also be used as edge devices.

This testbed will allow us to easily partition a model, deploy it and test various use cases, in such way that we can evaluate the inference latency and the network energy consumption under different partitioning and deployment strategies.

4.1 System Overview

The **DisRUPT** framework consists of four main components. The **MCU** Interface, the Raspberry Pi Interface, the Orchestrator Module, and the Partitioning Module.

The mentioned device interfaces components are responsible for the execution of the inference on the devices, as well as for the communication between them. They are automatically managed and the network performance is evaluated by the Orchestrator Module. While the Partitioning Module is responsible for partitioning the models, before the deployment stage, and it is not involved in the inference process itself.

To achieve the collaborative inference, we first use the Partitioning Module to partition the **DNN** models into all possible partitions that might be executed on the constrained devices. Then, the segments and devices that will execute them are selected according to the devices capabilities and the model requirements. Finally, the segments are deployed to the respective devices, that are also configured to communicate with each other effectively.

¹<https://www.raspberrypi.com/>

Each device in the network runs its corresponding interface, which is responsible for the execution of the assigned inference subtasks. Furthermore, it allows communication with other devices in the network, providing mechanisms to receive input data and transmit the results of the subtask.

In order to automate the deployment of the task to the network, we developed a set of scripts that not only create routes that chain together the inference pipeline, but also upload the partitioned submodels to the devices and feed the network with the input data. This results in the Orchestrator Module, that is given the network configuration and input data by the developer, through a configuration file. The Orchestrator Module is also capable of collecting timing and energy performance metrics during the inference process, for later analysis and comparison.

The interaction between the **DisRUPT** components can be visualized in Figure 4.1. Note that the inference interfaces can be scaled to accommodate more devices in the network.

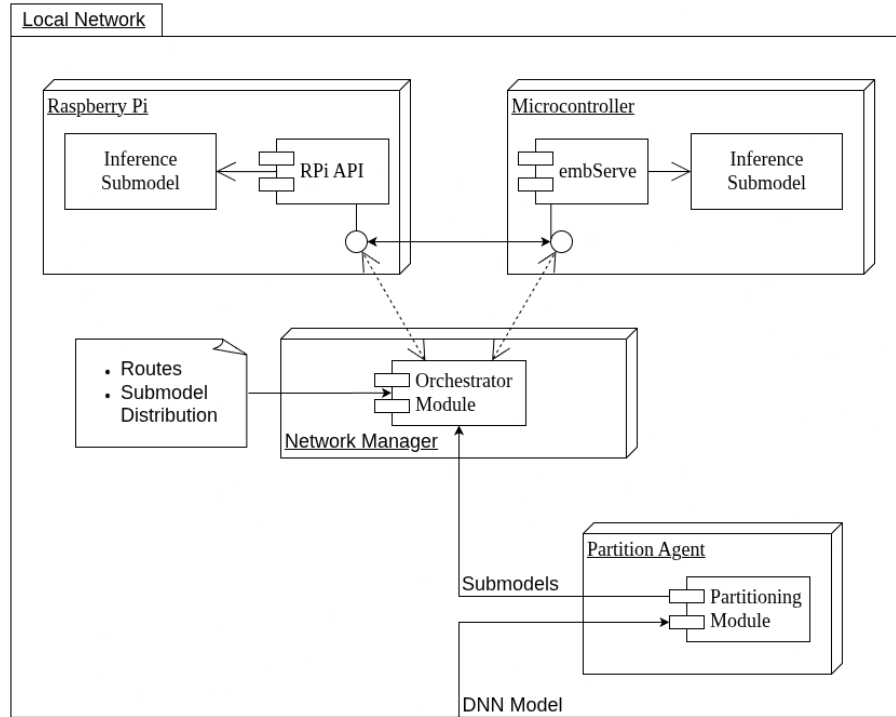


Figure 4.1: **DisRUPT** architecture.

4.2 Model Partitioning

Before the network configuration phase, we need to partition the **DNN** models into segments that are fit for the constrained devices in our network. To achieve this, we will use the Network-Aware Model Partitioning (**NAMP**)’s model surgery module, that allows us to easily partition and optimize Multilayer Perceptrons (**MLPs**) and **CNNs** into said segments. [8]

NAMP’s Partitioning Module takes a **DNN** as input and generates a set of all possible model partitions by breaking it down layer by layer. Additionally, certain Fully Connected (**FC**) layers can be partitioned to allow wide layers to be executed collaboratively across multiple devices.

This module also enables model quantization to optimize the resulting segments, reducing their memory footprint for deployment on resource-constrained devices.

To achieve this, the weights and activations of the submodels are quantized from 32-bit to 8-bit representations using TensorFlow’s post-training quantization tools. First, the module generates all possible submodels in the standard TensorFlow format for every combination of beginning and ending layers. If flagged for optimization, each submodel is then quantized to 8 bits and ultimately converted into the *.flite* format.

4.3 MCU Interface

To enable the execution of inference tasks on MCUs, we utilized embServe[17], a runtime application built on top of the Zephyr RTOS², which enables the deployment and execute services on constrained devices. This choice was motivated by our State of the Art analysis, as it fills the gap by supporting model deployment on far-edge devices with minimal overhead.

This framework provides a model agnostic interface to the underlying hardware, allowing us to easily deploy the partitioned submodels to the MCUs as services, and also to manage the communication routes between the devices in the network.

To deploy the partitioned submodels, we need to compile an embServe service: a C application that dictates the behavior of the device when that service is active, in this case, the inference execution upon receiving input data. The service is then uploaded to the device, and after defining the incoming and outgoing communication routes, the device is ready to partake in the inference network.

4.4 Raspberry Pi Interface

To facilitate the interaction with the Raspberry Pi and to enable the deployment of inference tasks, we have developed a custom API that allows us to manage the Raspberry Pi’s resources and execute inference tasks remotely, while also transmitting the intermediate results to the next device in the pipeline, monitoring the system along the way.

This was achieved by creating a Python runtime application on the Raspberry Pi. The application creates a server that listens for incoming requests to allow the Raspberry Pi to partake in the inference network, from reporting its status, having configurable outgoing communication routes, and executing the assigned inference tasks. It should be noted that the Python language does not hinder the performance of the inference tasks, as the used TFLite runtime is implemented in C, limiting the Python overhead to the communication parsing and the server management.

The server creates a set of objects, as described in Table 4.1, that interact with each other to allow for the integration of the device into the inference network, like we can see in Figure 4.2.

²<https://www.zephyrproject.org/>

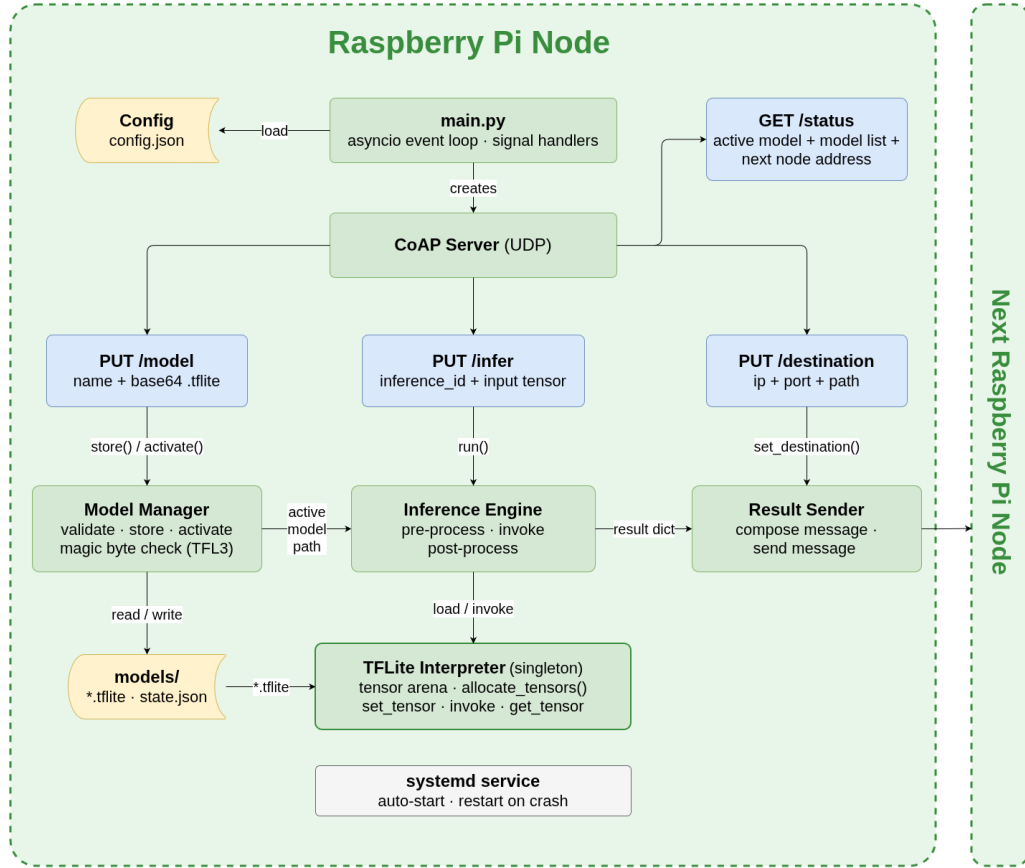


Figure 4.2: Raspberry Pi Interface architecture.

4.5 Deployment Planning

To reduce the impact of the limitations of the far-edge constrained devices on the performance of the distributed inference, we need to carefully plan the deployment and the communication between the devices during the inference process. This involves selecting the most suitable partitioning and deployment strategies, as well as optimizing the communication protocols to minimize latency and energy consumption.

In this section, we will discuss the strategies and techniques that can be employed to achieve the aforementioned goals, and how they can be implemented in the **DisRUPT** framework.

4.5.1 Device Communication

Besides each device being responsible for its assigned inference subtask, they also need to communicate with each other to bring meaning to the inference process, which takes a toll on the network performance. In order to ease the communication overhead between the devices, we have taken several measures such as the messaging protocol selection, the minimization of the data transmitted, and the reduction of the communication stages during the inference process.

Component	Type	Description
<code>main.py</code>	Entrypoint	Initializes all components, and runs the asyncio event loop.
Config	File	Stores runtime parameters including the network address, CoAP port, active model name, and destination routing.
CoAP Server	Object	Listens for incoming CoAP requests over UDP and dispatches them to the appropriate endpoint handler.
Model Manager	Object	Validates, stores, and activates <code>.tflite</code> model files, persisting the active model selection across restarts.
Inference Engine	Object	Manages the TFLite Interpreter lifecycle, pre-processes input tensors, invokes the model, and returns the output.
Result Sender	Object	Forwards inference results to the configured destination node via CoAP.
TFLite Interpreter	Object	Singleton instance of the TFLite runtime that allocates the tensor arena and executes the loaded model.
<code>GET /status</code>	Endpoint	Returns the currently active model and the configured destination address.
<code>PUT /model</code>	Endpoint	Accepts new <code>.tflite</code> files encoded in base64 to store or activate existing models.
<code>PUT /infer</code>	Endpoint	Accepts an input tensor and an inference identifier, runs inference, and forwards the result to the destination.
<code>PUT /destination</code>	Endpoint	Sets the IP address, port, and path of the next node in the inference pipeline.
<code>models/</code>	File directory	Contains all deployed <code>.tflite</code> model files and the state file tracking the active model.
systemd service	System service	Manages the runtime process, ensuring automatic startup on boot and restart on failure.

Table 4.1: Raspberry Pi Interface components.

The CoAP web transfer protocol [21] was selected for the communication between the devices, as it is a lightweight protocol, as presented in the Figure 4.3, designed for constrained devices and networks, making it well-suited for our use case. Specifically, CoAP operates over UDP rather than TCP, drastically reducing connection handshake overhead, transmission delays, and power consumption. To fit low-power payload limits, it uses a compact 4-byte binary header that reduces fragmentation. Its dual-layer design adds reliability via confirmable messages while preserving REST semantics (GET, POST, PUT, DELETE).

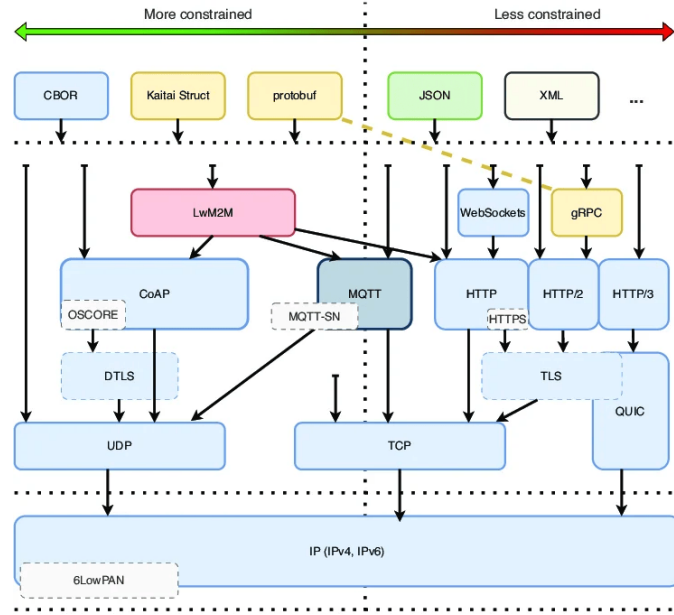


Figure 4.3: Communication protocol landscape for **IoT** (from Gulati et al. [9]).

To further minimize the communication overhead during inference, we have separated the communication into two main stages: the setup stage and the inference stage. During the setup stage, we transmit all the necessary information to allow the device to perform the inference task and to communicate with the other devices in the network, such as the routes used to manifest the inference pipeline, as well as the services containing the inference model logic. On the other hand, during the inference stage, we restrain the communication to only the necessary information for the inference process, which is the intermediate results between the segments. For inter-device communication, we encode payloads in Concise Binary Object Representation (**CBOR**) [4], a compact binary serialization format. Compared with text-based JavaScript Object Notation (**JSON**), **CBOR** reduces message size and parsing cost. We use **JSON** for setup and debugging (when messages are inspected by tools or developers), but avoid it during production inference.

The full structure of the messages exchanged between the devices during the inference stage is presented in Listing 4.1. Including the wrapper that allows the compatibility with embServe’s interface, and the nested structure that contains data used to identify the inference task and collect the performance metrics, namely the inference latency.

Listing 4.1: Inter-device message structure during the inference stage.

```

1 struct MessageData {
2     uint8    inference_id;           // 1 Byte : Orchestrator's Inference Task ID
3     uint32    inference_latency_ms;  // 4 Bytes : Inference latency in milliseconds
4     uint8[]   input_data;           // Variable: Dynamic inference input payload
5 };
6
7 struct Message {
8     uint8      service_id;           // 1 Byte : EmbServe service identifier
9     char[]     key_input;            // Variable: EmbServe service endpoint key
10    MessageData message_data;        // Variable: Nested MessageData structure
11 };

```

4.5.2 Performance Metrics

One of the goals of this work is to evaluate the performance of the inference network under different partitioning and deployment strategies. With **DisRUPT**, performance can be measured through the inference latency and the network energy consumption, which are crucial indicators of the inference process efficiency and effectiveness.

The Orchestrator Module is responsible for collecting these performance metrics during the inference process. In particular, the inference latency is measured by recording the timestamp of a given inference task and matching it with the timestamp of the final result using a unique identifier assigned to each distributed inference task. The inference task unique identifier is only 1-byte long, allowing for a maximum of 256 concurrent inference tasks.

To separate the inference latency from the communication overhead, we also measure the latency of each segment execution on each device, in a similar method. This is achieved by transmitting the sum of the previous segments execution time along with the in-device latency measurement.

The energy consumption of the network is measured using an external power profiler connected to the devices, allowing us to monitor the energy usage during the inference process and reporting it to the Orchestrator Module, as to not interfere with the devices' performance. This data is stored for further analysis and comparison between different strategies.

All this data allows us to estimate the performance of the inference network under different configurations, and to identify the most effective strategies for deploying **DNN** models on constrained devices. This data can also be used to develop several heuristics for future deployments, allowing us to match the user needs with the most suitable partitioning and deployment strategies, to achieve minimal latency, energy consumption, or a balance between the two.

4.5.3 Partition and Deployment Strategies

Another important aspect of the deployment planning is the selection of the partitioning and deployment strategies. The way we partition the model and assign the segments to the devices can

have a significant impact on the performance of the inference network.

When it comes to the deployment strategy, the most relevant aspect to consider is the **DNN** architecture, as some layers are more computationally intensive than others. Even if the layers in **CNN** models and **MLP** models are based on a series of matrix multiplications, the variations in the dimensions of the matrices and the number of parameters can lead to significant differences in the computational load of each layer.

In order to achieve the best performance, we need to carefully select the devices that are more appropriate for the execution of each segment, considering their computational capabilities and the memory constraints.

Nonetheless, the relationship between the layers' computational load and the devices' capabilities is not always straightforward, so for this work, we will create several scenarios with different deployment strategies, to evaluate their performance and to identify the most effective ones for different use cases and requirements, as presented in Table 4.2, while always considering the constraints of the devices in the network. Faced with several possibilities for a given deployment strategy, we will select the one that keeps the model architecture intact, as to keep together layers that belong to the same block.

Strategy	Description	Reasoning
Edge Only	The full model is executed on just the edge device.	This can be used as a baseline for comparison with other strategies.
Edge First	The model is partitioned to execute the first segments on the edge device, and the remaining segments on the MCU devices.	For some models, the first layers are more memory intensive, and therefore more suitable for execution on the edge device, while the remaining layers can be executed on the MCU devices.
Edge Last	The model is partitioned to execute the last segments on the edge device, and the first segments on the MCU devices.	For some models, the last layers are more computationally intensive, and therefore more suitable for execution on the edge device, while the first layers can be executed on the MCU devices.
Min-Cut	The model is partitioned to minimize the communication cost between devices, by selecting partition points where the Intermediate Tensors are smallest.	This strategy aims to reduce the transmission cost between devices, thus shortening the overhead of collaborative inference.
Load-Balanced	Segments are distributed across the maximum number of devices to minimize the computational load on each, taking into account both the layers' computational load and the devices' capabilities.	For this strategy, we balanced the memory footprint of each segment across devices, to avoid bottlenecks and to maximize the utilization of the available resources.
Greedy	Partition points are selected to fit as much of the model as possible onto each MCU device.	This strategy aims to maximize the utilization of each device's resources, potentially reducing the number of devices needed for inference.

Table 4.2: Deployment strategies for model partitioning and segment assignment.

Chapter 5

Experiments and Results

In this chapter, we present the core segment of this work. Firstly, the experimental setup is described, detailing the network topology, the energy measurement process, and the models used for validation of our framework as well as to gather the data for the results analysis. Then, the performance metrics of the different partitioning strategies and distribution configurations are presented and analyzed, with a focus on the trade-offs between latency and energy consumption, aiming to profile the impacts of these strategies on the model inference and understand the potential benefits of collaborative inference in edge and far-edge devices.

5.1 Experimental Setup

In an effort to validate our framework and also to understand the impact of partitioning and distributing the inference process across multiple devices, we conducted a series of experiments. The experiments were designed to evaluate the performance, in terms of latency and energy consumption, of different partitioning strategies and distribution configurations across a set of models with varying complexity and architecture.

5.1.1 Network Topology

To put the framework into practice, we set up a local network consisting of four Particle Argon¹ microcontrollers and a single Raspberry Pi 3 Model B², all connected via Wi-Fi. This network is controlled and monitored by an external computer, which is responsible for orchestrating the experiments, collecting data, and processing results.

The choice of using a single Raspberry Pi was motivated by the need to have a more powerful device to handle the inference tasks of more complex models, expanding the range of models our framework can support and evaluate. At the same time, we wanted to maintain a balance and mitigate the bias toward a Raspberry Pi-centric execution for all experiments.

¹<https://docs.particle.io/argon/>

²<https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>

This network structure allows us to better understand the trade-offs and possible benefits of having **MCUs** in the network, and take advantage of their potential in other real-life heterogeneous networks, as it keeps a balance between higher performant and power consuming devices, and lower performant and power efficient devices.

For the **MCU** devices, we emulated the execution of each possible partition to determine the *Arena Size*, the memory required for all the intermediate tensors, from the input to the output, to execute the segment. Looking further into the used embServe service’s template and **MCU** image, we could establish a lower bound for the memory footprint of each segment, as presented in Equation 5.1. Not only, the size of the buffers used to store the input and output tensors, but also the submodel as a whole. The *InputSize* is accounted for twice, as the input data needs to be stored in the communication buffer, and then copied to the service input buffer.

$$\text{Memory Footprint} = 2 \times \text{Input Size} + \text{Output Size} + \text{Arena Size} + \text{Model Size} \quad (5.1)$$

To aid on the analysis of the memory footprint of each submodel, we created a table for each model, like the one presented in Figure 5.2. In these tables, rows represent the starting layer of the submodel and columns represent the ending layer of the submodel. Each cell in the table represents the corresponding submodel, and is color-coded based on its memory footprint. Yellow cells represent a memory footprint above 55 kB, but below 80 kB, which is the maximum available memory for the Particle Argon **MCU**, after accounting for the memory used by the operating system and embServe. Similarly, red cells represent a memory footprint above the 80 kB limit.

5.1.2 Energy Measurement

In addition, we connected an Argon **MCU** to a Joy-IT USB-Multimeter 120³, a device that reports a variety of electrical parameters, including voltage, current, and power consumption in real time. Sampling the circuit at a 10 ms interval, with an accuracy of five decimal places.

We ran each submodel that participated in the inferences included in the experiments, and collected the data from the multimeter, which allowed us to profile the energy consumption of each submodel, and, by extension, the energy consumption of the entire inference process. This process was repeated for the Raspberry Pi, to collect data on the energy consumption of the submodels that were deployed on it. The results of this process are presented in Appendix B, where we present the full energy measurements of each submodel.

To isolate the energy consumption of the inference process, we also measured the idle consumption of the devices, that is, the energy consumed by the devices when they are not performing any user task, and subtracted the corresponding energy value from the total energy consumption. For the Argon **MCU**, the idle consumption was measured to be 0.135026 W, while for the Raspberry Pi, the idle consumption was measured to be 1.271197 W.

³<https://joy-it.net/en/products/JT-UM120>

5.2 Models

To validate our framework, we selected a set of models that are representative of the edge computing domain, with varying levels of complexity and architecture. These models try to cover a wide range of applications and use cases, from simple image classification tasks to keyword and anomaly detection. All this while keeping a focus on model size for the constraints of the devices on the edge and far-edge. The models we selected, partitioned, distributed across the network, and had the performance evaluated will be described in the following subsections. A global overview of the layer-by-layer information for each model is provided in Appendix A.

5.2.1 All-CNN-C

The first model we present is the All-CNN-C model [23], an image classification model selected for its architectural simplicity and efficiency. This decision was primarily motivated by the fact that it is a simple CNN with the peculiarity of being composed entirely of convolutional layers, replacing the traditionally used pooling and fully connected layers with strided convolutions. This makes it a good candidate, as the layer types are similar to each other, making the computational load uniform along the network, and, therefore, allowing for a large amount of possible balanced partitions to distribute.

5.2.1.1 Model Architecture

The All-CNN-C architecture is characterized by its uniform application of convolutional layers. Unlike standard architectures that reduce spatial resolution via max-pooling, All-CNN-C utilizes strided convolutions to downsample the input.

The network comprises nine convolutional layers, followed by a global average pooling layer. Specifically, the architecture applies a series of 3×3 convolutions, with specific layers utilizing a stride of 2 to reduce spatial dimensions. The network concludes with a global average pooling layer that maps the resulting feature maps into a single vector, which is then passed to a softmax layer for classification, as represented in Figure 5.1. It should be noted that, to further compact the model to fit the devices constraints, we used a width multiplier of $\alpha = 0.125$, that limit the number of filters in each convolution, and, therefore, the size of intermediate feature maps.

This design effectively eliminates the parameter-heavy fully connected layers, significantly reducing the model's memory footprint while maintaining competitive accuracy on benchmarks such as CIFAR-10.

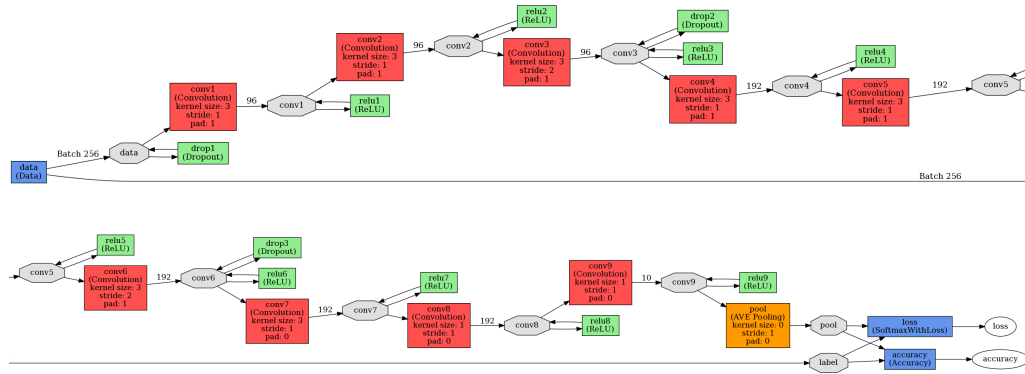


Figure 5.1: All-CNN-C architecture diagram from Buda [5].

5.2.1.2 Partitioning Strategies

After partitioning the model into a set of all possible partitions, from any beginning layer to any ending layer, we arrived at a total of 66 submodels, which were then emulated and had the lower bound for their memory footprint calculated, as described in Figure 5.2.

		Target Layer (j)											
		0	1	2	3	4	5	6	7	8	9	10	11
Source Layer (i)	0	-	39.2 KB	52.2 KB	44.8 KB	52.4 KB	59.6 KB	61.7 KB	68.0 KB	70.3 KB	71.1 KB	71.2 KB	71.7 KB
	1	-	-	70.5 KB	63.1 KB	70.7 KB	78.0 KB	80.0 KB	86.3 KB	88.6 KB	89.5 KB	89.5 KB	90.0 KB
	2	-	-	-	50.7 KB	58.3 KB	65.3 KB	67.6 KB	73.9 KB	76.2 KB	77.1 KB	77.1 KB	77.6 KB
	3	-	-	-	-	29.4 KB	40.2 KB	42.5 KB	48.8 KB	50.8 KB	51.7 KB	52.0 KB	52.2 KB
	4	-	-	-	-	-	42.6 KB	44.6 KB	51.2 KB	53.2 KB	54.1 KB	54.4 KB	54.6 KB
	5	-	-	-	-	-	-	32.7 KB	39.0 KB	41.3 KB	42.1 KB	42.2 KB	42.7 KB
	6	-	-	-	-	-	-	-	16.3 KB	18.3 KB	19.5 KB	19.5 KB	19.8 KB
	7	-	-	-	-	-	-	-	-	8.3 KB	9.1 KB	9.4 KB	9.7 KB
	8	-	-	-	-	-	-	-	-	-	6.4 KB	6.4 KB	7.0 KB
	9	-	-	-	-	-	-	-	-	-	-	3.1 KB	3.4 KB
	10	-	-	-	-	-	-	-	-	-	-	-	2.9 KB

Figure 5.2: All-CNN-C submodel memory footprints.

According to the deployment strategies established in Chapter 4, we defined the following distribution strategies for the All-CNN-C model, as per represented in Table 5.1.

Strategy	Selected Submodels	RPi Assignment
Edge Only	0-11	0-11
Edge First	0-8 + 8-11	0-8
Edge Last	0-3 + 3-11	3-11
Min-Cut	0-3 + 3-7 + 7-11	3-7
Load-Balanced	0-3 + 3-6 + 6-9 + 9-11	N/A
Greedy	0-3 + 3-6 + 6-11	N/A

Table 5.1: All-CNN-C distribution configuration.

5.2.1.3 Result Analysis

After executing a batch of 255 inference tasks for each distribution strategy, as the unique identifier is only 1-byte long, we collected the following results, as represented in Figure 5.3 and Figure 5.4. We can take note that the Edge Only strategy dominates in terms of latency, as its average latency is almost 25% of the second fastest strategy, Edge First, and almost 2.5% of the slowest strategy, Load-Balanced.

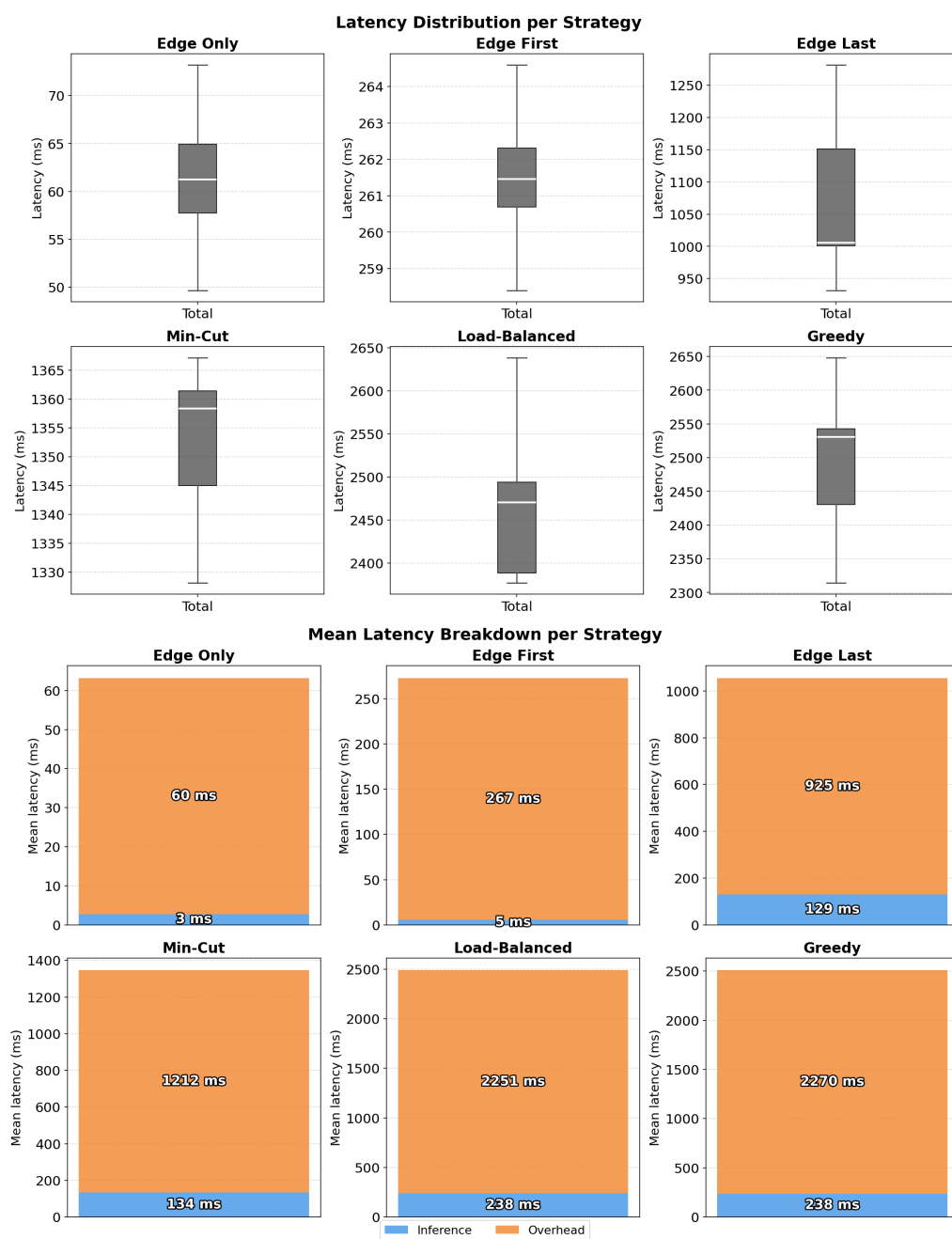


Figure 5.3: All-CNN-C inference latency measurements.

In terms of energy consumption, as represented in Figure 5.4, we can observe that the Edge Only strategy is the most energy efficient, with an average energy consumption of 56.6 mJ, which is almost 2 times less than the second most energy efficient strategy, Edge First.

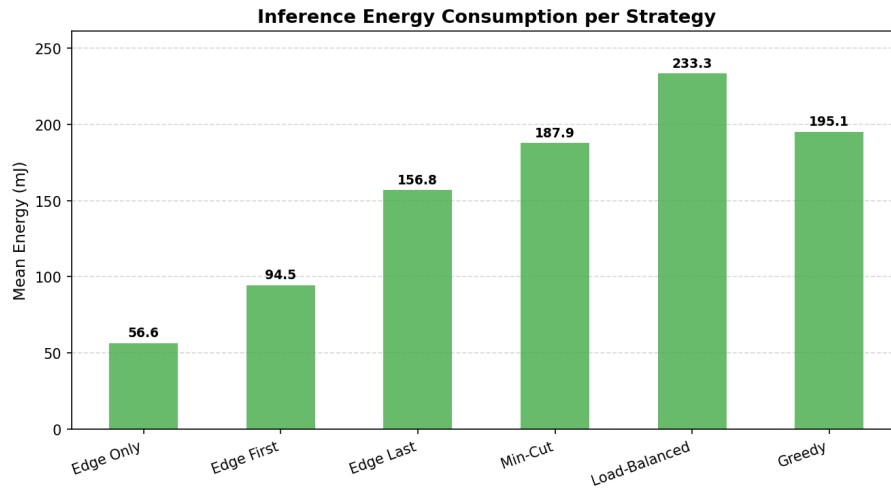


Figure 5.4: All-CNN-C energy consumption measurements.

Overall, the results indicate that the Edge Only strategy is the most efficient in both the analysed metrics. However, it is important to note these metrics are only accounting for the inference process, and not the idle periods of the devices, which can be significant in a real-world scenario, and turn the Edge Only strategy into a less efficient strategy, as the Raspberry Pi has a higher idle power consumption.

5.2.2 DS-CNN

The second architecture used to evaluate the performance of different heuristics is the Depthwise Separable Convolutional Neural Network (**DS-CNN**) model [29], which is a highly optimized, lightweight convolutional neural network designed for efficient keyword spotting and localized inference on resource-constrained devices, such as far-edge devices. This model shifts away from traditional computationally heavy 3D convolutions by uncoupling spatial and channel-wise feature extraction, which significantly reduce the number of parameters and computational cost in each convolutional layer. This makes **DS-CNN** an exemplary model for TinyML, as it can be flexibly partitioned and distributed across a given network of devices without having its evaluation performance compromised.

5.2.2.1 Model Architecture

The model configuration utilized in our evaluation corresponds to the **DS-CNN** delineated in the core *Hello Edge* [29] framework, with a width multiplier of $\alpha = 0.25$, to further help us fit the model to the constraints of far-edge devices.

The network architecture, represented in Figure 5.5, initiates with a standard 2D convolutional layer, which acts as a primary feature processor over the input time-frequency spectrogram data. This baseline layer prepares the multi-channel feature representation by mapping local correlations across a predefined set of feature channels.

Following the initial standard convolution, the core feature extraction pipeline is composed of four sequential depthwise separable convolutional blocks. Each block factorizes the standard convolution mechanism into two isolated operations:

- **Depthwise Convolution:** Applies a dedicated 2D spatial filter per individual input channel independently to capture localized temporal and spectral features.
- **Pointwise Convolution:** Implements a 1×1 convolution across the full channel depth, linearly combining the separate outputs of the depthwise phase and projecting them into a brand new channel space.

Every independent convolutional operation within these blocks is structurally bound to a subsequent Batch Normalization layer followed by a Rectified Linear Unit (ReLU) activation function, ensuring robust optimization stability during the forward pass. The network is concluded by a global average pooling layer that compresses the dimensional spatial map into a singular feature vector, which is subsequently routed through a fully connected softmax layer to compute the localized classification probabilities.

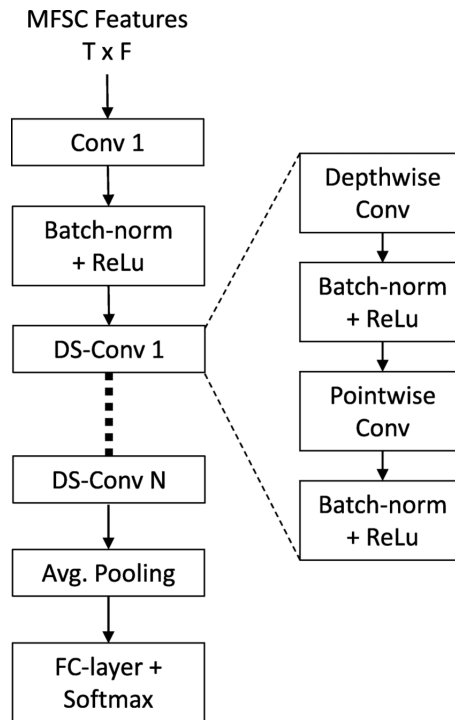


Figure 5.5: DS-CNN architecture diagram from Sørensen, Epp, and May [22].

5.2.2.2 Partitioning Strategies

We obtained a total of 435 submodels from the partitioning of the **DS-CNN** model, which were then emulated and had their memory footprint profiled, as represented in Figure 5.6.

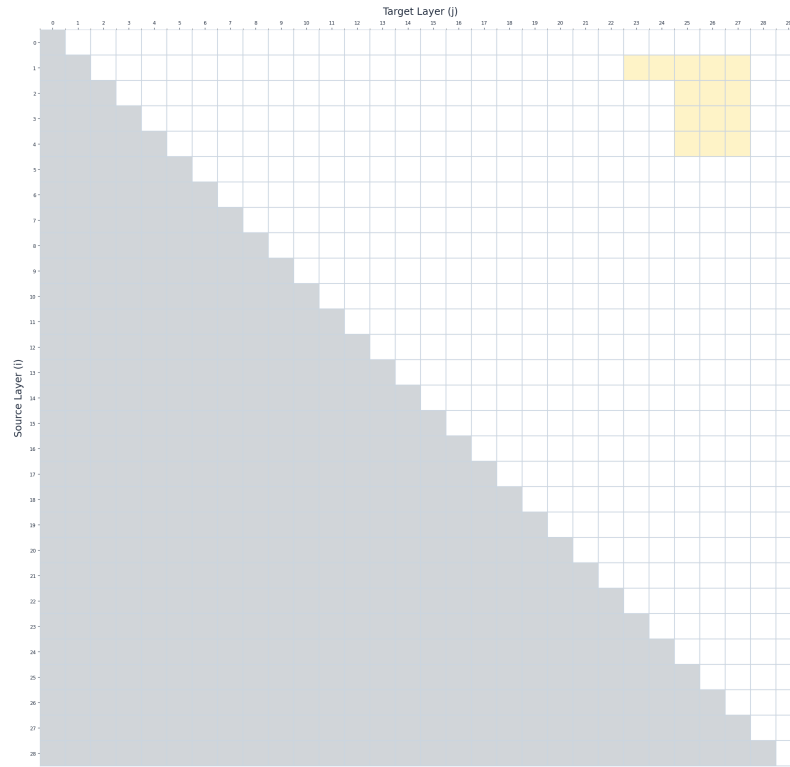


Figure 5.6: DS-CNN submodel memory footprints.

From this set of submodels, we defined the distribution strategies for the **DS-CNN** model to be as represented in Table 5.2.

This model has the particularity that every layer, with exception of the last one, have the same output shape. This fact renders the Min-Cut strategy ineffective, as following it will not result in a new partition.

Strategy	Selected Submodels	RPi Assignment
Edge Only	0-29	0-29
Edge First	0-15 + 15-21 + 21-29	0-15
Edge Last	0-9 + 9-15 + 15-29	15-29
Load-Balanced	0-9 + 9-15 + 15-21 + 21-29	N/A
Greedy	0-29	N/A

Table 5.2: DS-CNN distribution configuration.

5.2.2.3 Result Analysis

The 255 inference batch executed for each distribution strategy yielded the results presented in Figure 5.7 and Figure 5.8. Regarding latency, the Edge Only strategy once again demonstrates superior performance, by a large factor.

We highlight the delay that the overhead latency introduces in the overall latency of the distributed inference process. Since the submodels output a large intermediate tensor, the communication overhead is evident. The strategies that rely on fewer submodels, and therefore, fewer communication steps, are the ones that perform better in terms of latency.

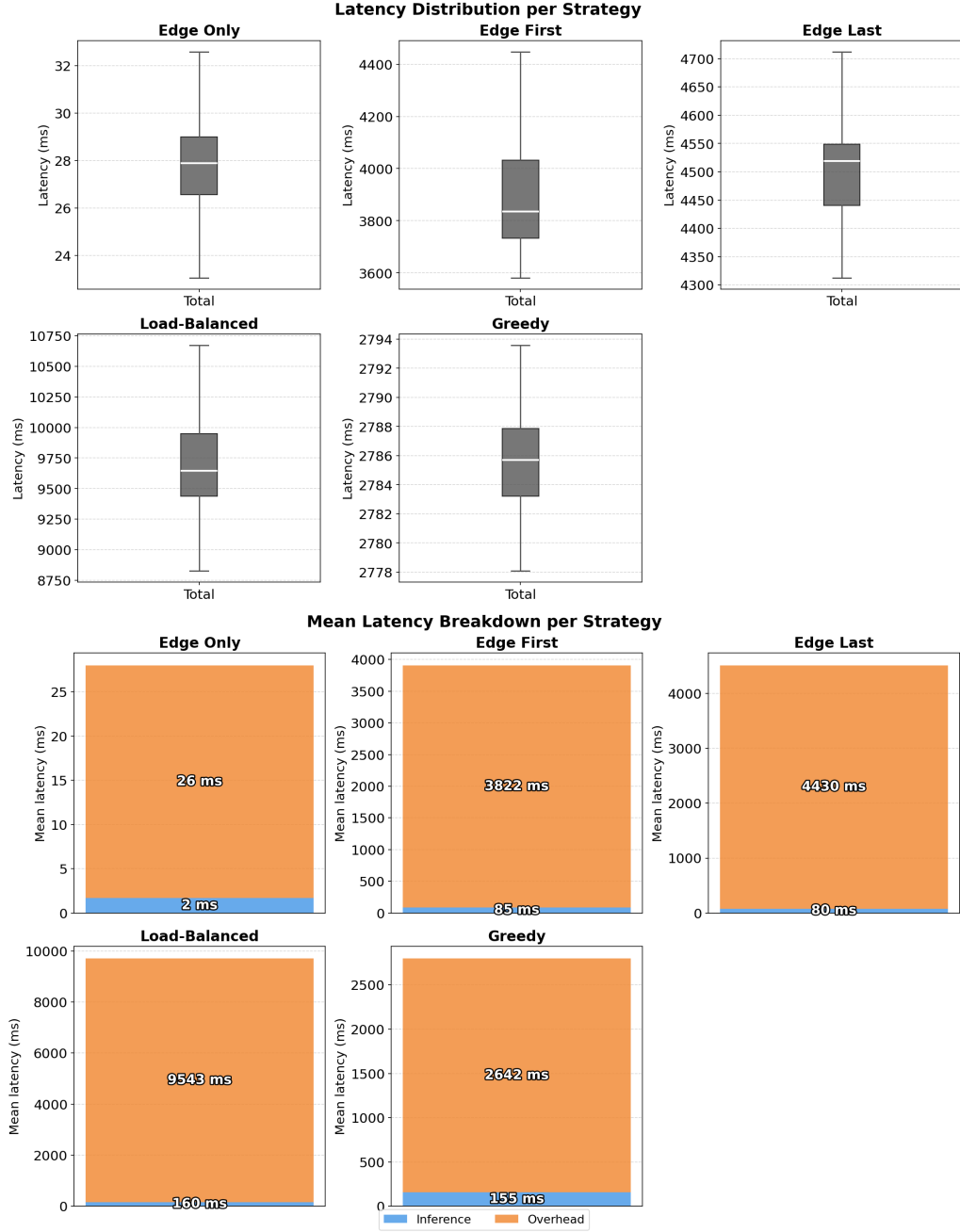


Figure 5.7: DS-CNN inference latency measurements.

Without the communication overhead that the other strategies inherit, the Greedy strategy outperforms the Edge Only strategy in energy consumption, regardless of the higher latency.

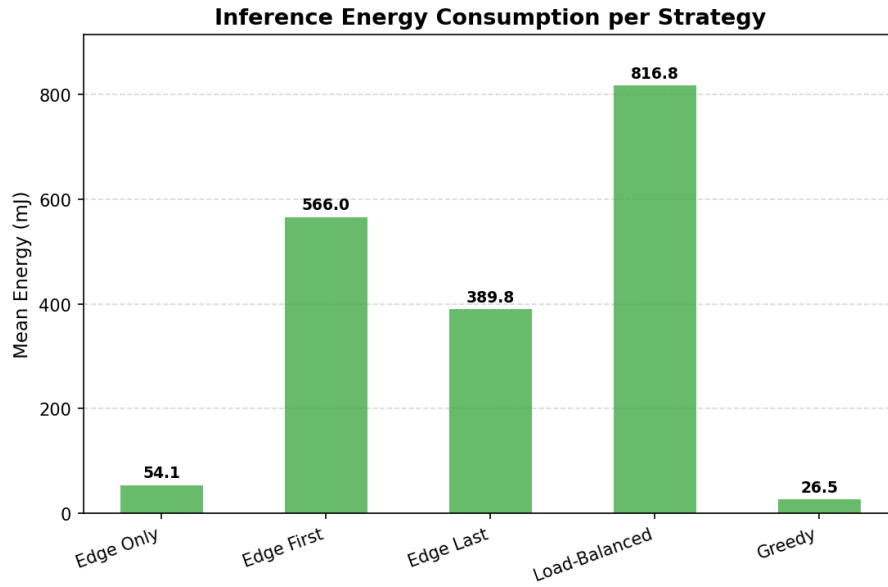


Figure 5.8: DS-CNN energy consumption measurements.

For this model, we can conclude that the Edge Only strategy is the most appropriate if the priority is to minimize latency, while the Greedy strategy is the most fit to minimize energy consumption, at the cost of a higher latency. No strategy is able to achieve a balance between the two metrics with distinction.

The model constant intermediate tensor sizes throughout the network also impact the results, as we can notice on the similarity of the metric on the Edge First and Edge Last strategies, which have the same number of submodels, communication steps, and number of layers per device.

5.2.3 MobileNetV1

The third and most complex model we present is the MobileNetV1[12], a highly popular architecture in the mobile and edge computing domain designed for image classification and object detection, among other computer vision tasks, while maintaining a low computational footprint. This model poses a significant challenge, by its sheer size and complexity, but also presents itself as an opportunity to collect data closer to a real-life scenario, since this model is widely used and integrated in a variety of applications and frameworks.

5.2.3.1 Model Architecture

MobileNetV1 is built upon the concept of depthwise separable convolutions, as previously described in section 5.2.2.1. However, it introduces a more complex, modular structure, utilizing 82 total layers. Unlike the static blocks in DS-CNN, MobileNetV1 employs a highly efficient architecture that facilitates hyperparameter tuning through width and resolution multipliers, allowing the model to be scaled according to specific latency or accuracy constraints.

The architecture relies on several depthwise separable blocks, as illustrated in Figure 5.9, where a depthwise convolution is followed by a pointwise convolution, both bundled with Batch Normalization and ReLU6 activations. This structure provides a significantly deeper feature extraction path, making it an ideal candidate for evaluating how our partitioning framework handles long, complex computational chains across heterogeneous edge devices.

In our evaluation, we utilize the MobileNetV1 with a width multiplier of $\alpha = 0.25$, an input resolution of $32 \times 32 \times 3$, and a final output layer of 256 classes. Additionally, we also cropped the model to remove the last two blocks, as they represented a challenge to the memory constraints of the far-edge devices, and would have further limited the number of possible partitions to distribute across the network.

This approach was chosen to maintain a balance between model complexity and the constraints of our far-edge devices, while still providing a realistic scenario for tests.

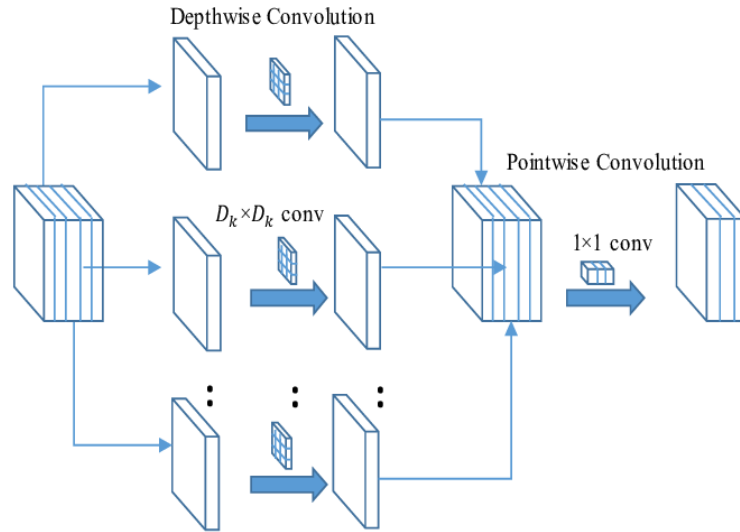


Figure 5.9: MobileNetV1 depthwise separable convolution structure from Nguyen [16].

5.2.3.2 Partitioning Strategies

The pruned MobileNetV1 model was partitioned into a total of 2568 submodels. However, the classifier head raised a challenge to the memory constraints of the far-edge devices, even if reduced to a minimum size, as indicated through the emulator results, presented in Figure 5.10.

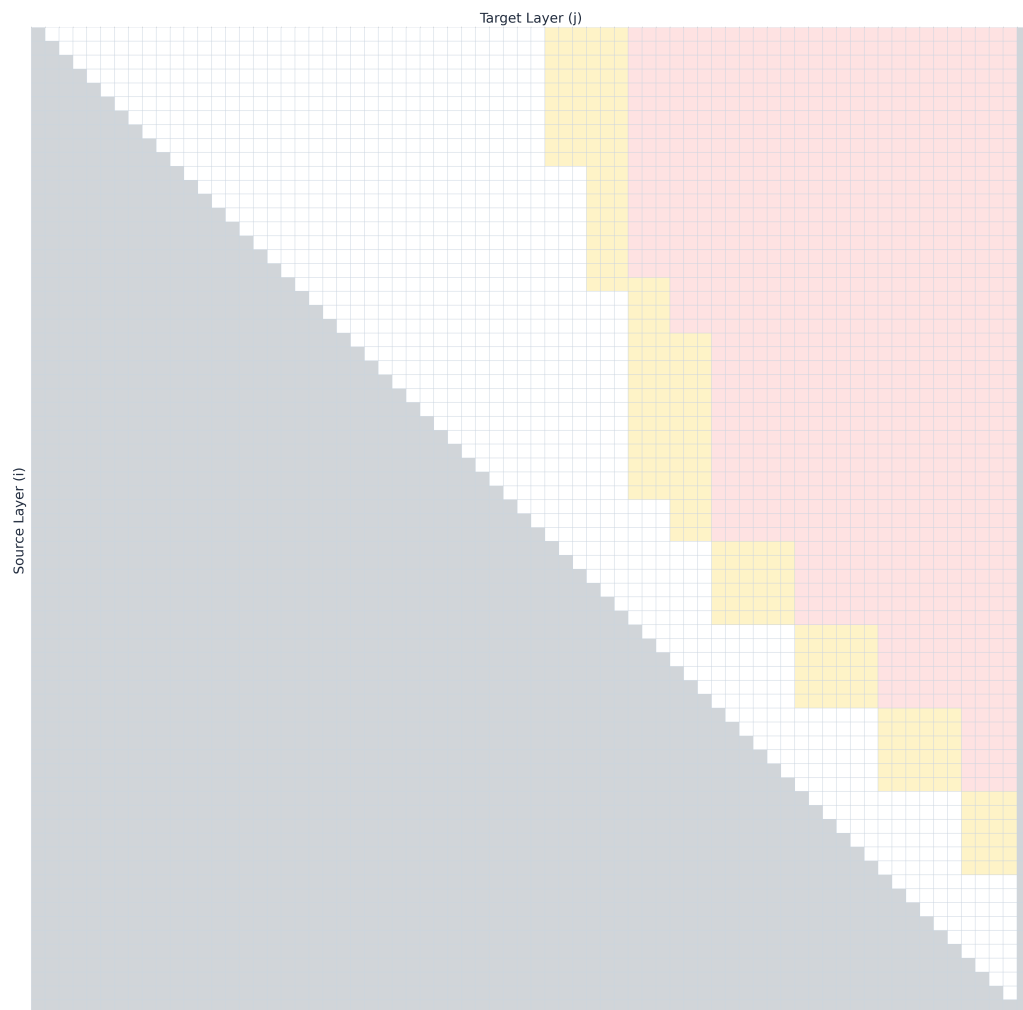


Figure 5.10: MobileNetV1 submodel memory footprints.

The model computational demand led to a limitation in the number of strategies that we could evaluate, as the final layer had always to be executed on the Raspberry Pi. This condition also made so most of the strategies resulted in the same distribution, therefore, we only present the strategies that resulted in different distributions, as represented in Table 5.3.

Strategy	Selected Submodels	RPi Assignment
Edge Only	0-71	0-71
Edge Last	0-15 + 15-27 + 27-39 + 39-71	39-71
Min-Cut	0-24 + 24-36 + 36-71	36-71

Table 5.3: MobileNetV1 distribution configuration.

5.2.3.3 Result Analysis

Executing the same batch of 255 inference tasks for each distribution strategy, we collected the results presented in Figure 5.11 and Figure 5.12.

In terms of latency, the presented strategies repeat the pattern observed in the previous models, where the Edge Only strategy is the fastest, followed by the Min-Cut strategy, and the Edge Last strategy falling behind.

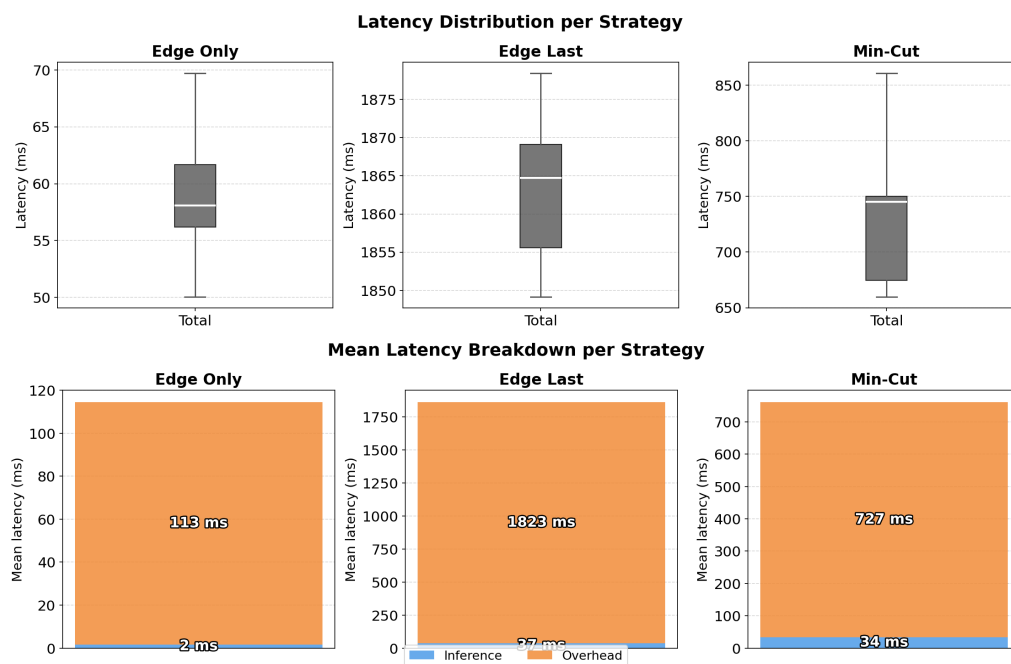


Figure 5.11: MobileNetV1 inference latency measurements.

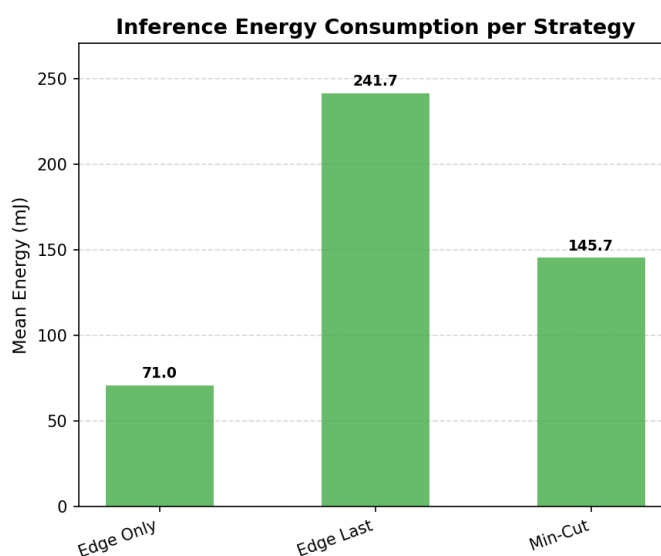


Figure 5.12: MobileNetV1 energy consumption measurements.

Moreover, this model also corroborates the findings from the previous models, in terms of energy consumption, where the Edge Only presents itself as the most energy efficient, but not significantly more than the other strategies. This is especially noticeable in this model, as the strategies rely on a similar number of devices.

In particular, it is remarkable that the Edge Last strategy, although it has the same number of submodels and communication steps as the Min-Cut strategy, it is less energy efficient than the latter. This indicates that selecting partition points based on the size of the intermediate tensors, as the Min-Cut strategy does, can lead to a more energy efficient process than simply relying on the higher computational capabilities of the Raspberry Pi, unlike the Edge Only strategy indicates.

5.2.4 MLP Autoencoder

The last model we present is a simple **MLP** autoencoder [13], fit for anomaly detection. This model was selected for its almost archetypal architecture, which allows us to evaluate the performance of the partitioning and distribution strategies in the most simple scenario, which is often used in more complex models as a building block.

5.2.4.1 Model Architecture

The autoencoder model is a **MLP** composed of a simple feedforward architecture, with an input layer, a set of hidden layers, and an output layer.

The initial layers are designed to process the data and extract relevant features, encoding them into a lower-dimensional, latent space. The subsequent layers decode this representation back into the original input space, aiming to reconstruct the input data as accurately as possible, as represented in Figure 5.13.

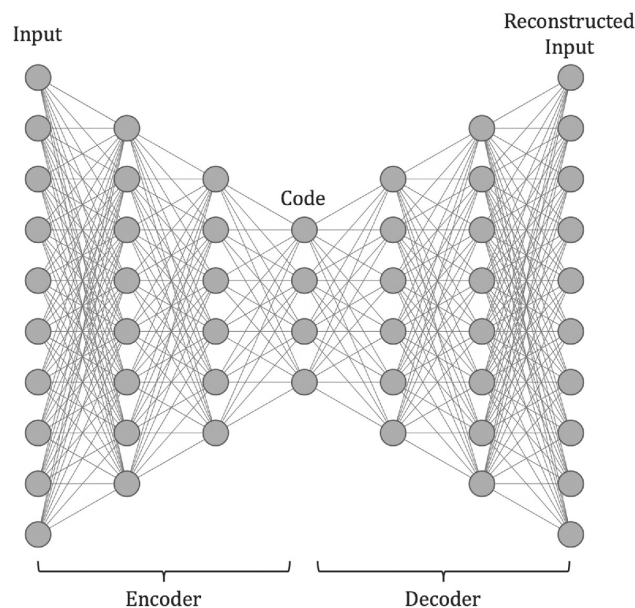


Figure 5.13: Autoencoder MLP architecture from Halder, Fidkowski, and Maki [10].

The model is trained to minimize the reconstruction error, learning to reconstruct the normal patterns by training the model using normal data. Therefore, it can be used to detect anomalies by identifying instances where the reconstruction error exceeds a certain threshold.

In particular, this model adaptation is composed of an encoder of four blocks, with 96 neurons each, a middle block composed of 8 neurons, a decoder of four blocks, similar to the encoder, with 96 neurons each, and an output layer with 96 neurons. Each mentioned block is composed of a dense layer, followed by a batch normalization layer and an activation layer, with the exception of the output layer, which does not have an activation layer.

5.2.4.2 Partitioning Strategies

The partitioning of the MLP Autoencoder model resulted in a total of 406 submodels, which can be seen in Figure 5.14. The model features a highly symmetric architecture, with the encoder and decoder being mirror images of each other, which results in a similar memory footprint for the submodels that are symmetric around the middle block.

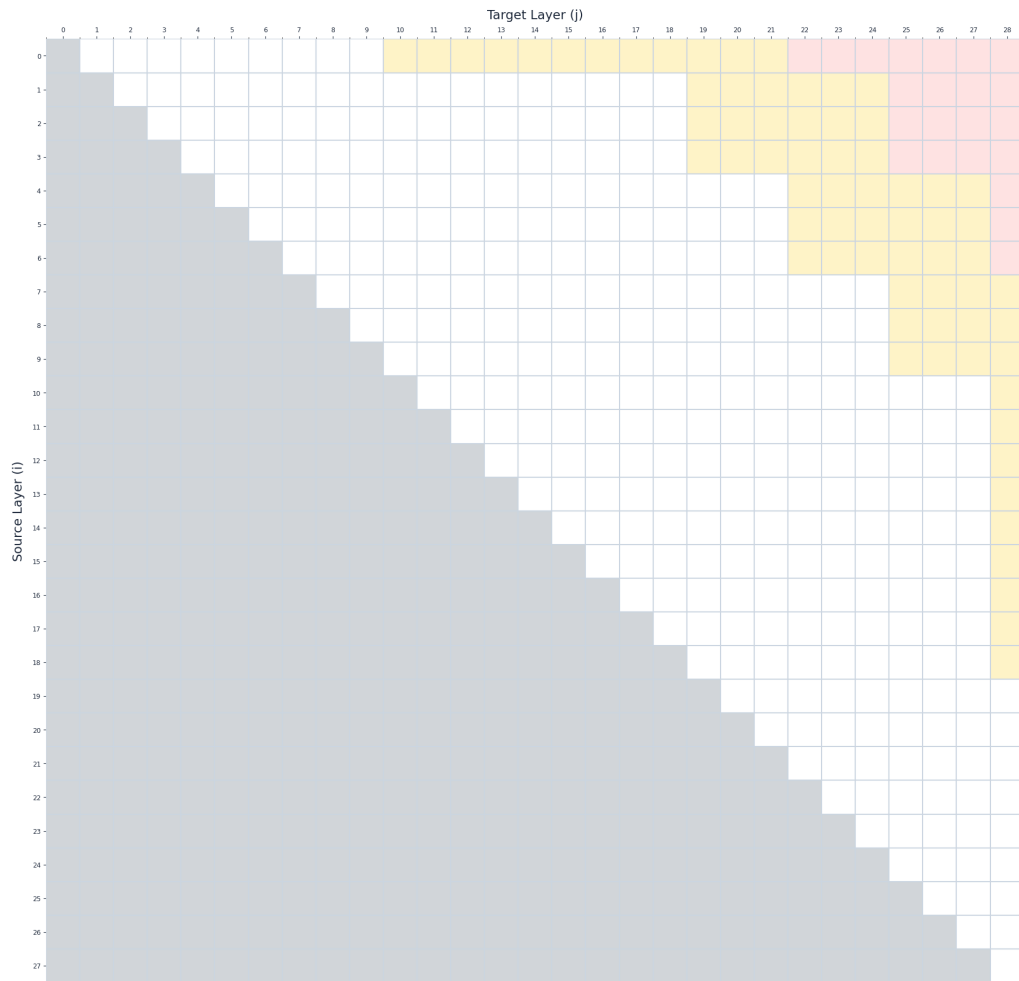


Figure 5.14: MLP Autoencoder submodel memory footprints.

The distribution strategies for the MLP Autoencoder model are represented in Table 5.4. The Greedy strategy was not included in the evaluation, as it would result in the same distribution as the Min-Cut strategy, since no feasible partition distribution could result in a lower number of MCUs devices used.

Strategy	Selected Submodels	RPi Assignment
Edge Only	0-28	0-28
Edge First	0-18 + 18-24 + 24-28	0-18
Edge Last	0-3 + 3-9 + 9-28	9-28
Min-Cut	0-6 + 6-24 + 24-28	6-24
Load-Balanced	0-6 + 6-18 + 18-24 + 24-28	N/A

Table 5.4: MLP Autoencoder distribution configuration.

5.2.4.3 Result Analysis

Running the 255 inference batch for each distribution strategy, the results presented in Figure 5.15 and Figure 5.16 were collected.

The model symmetry is reflected in the results, as the Edge First and Edge Last strategies have close results in both latency and energy consumption.

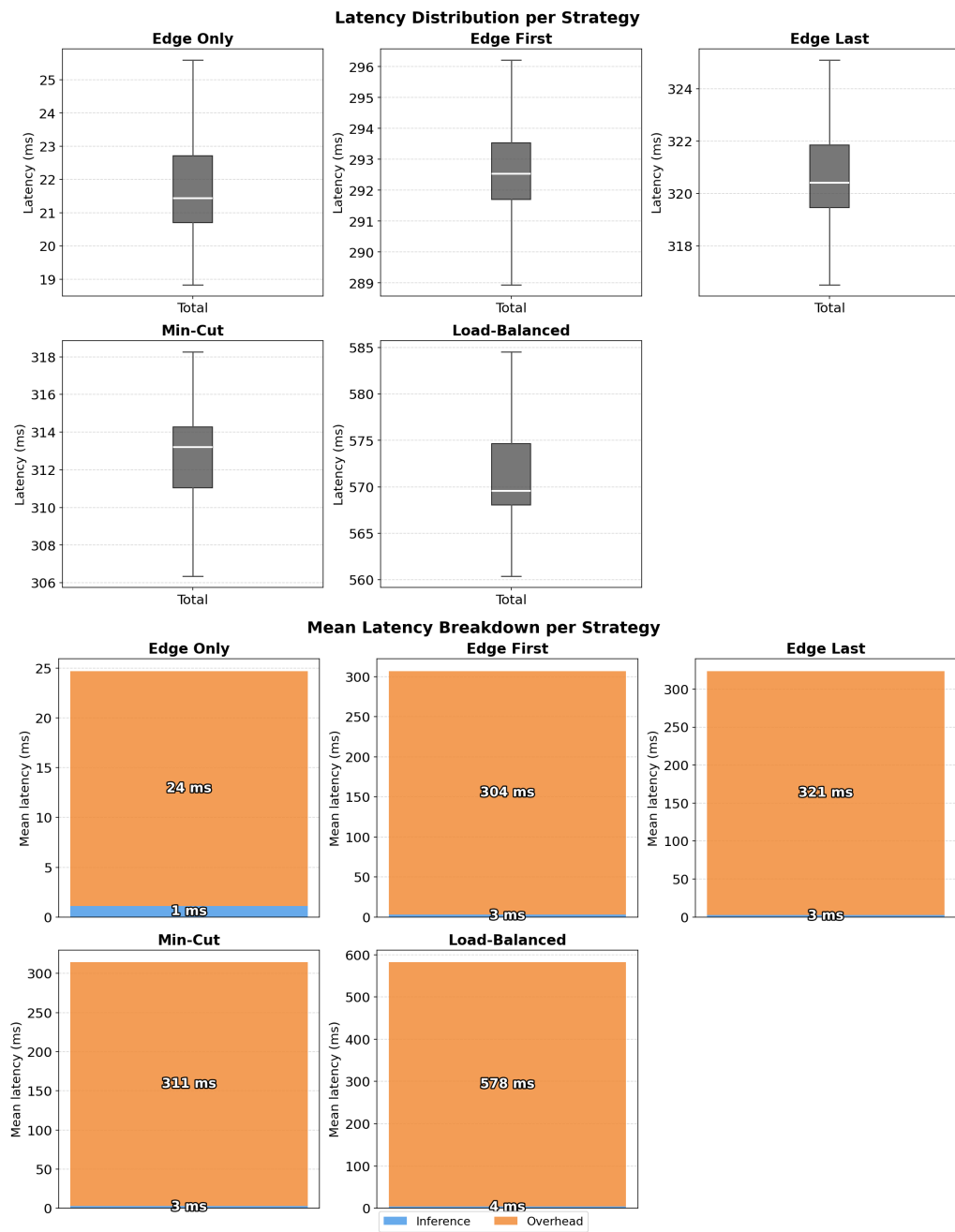


Figure 5.15: MLP Autoencoder inference latency measurements.

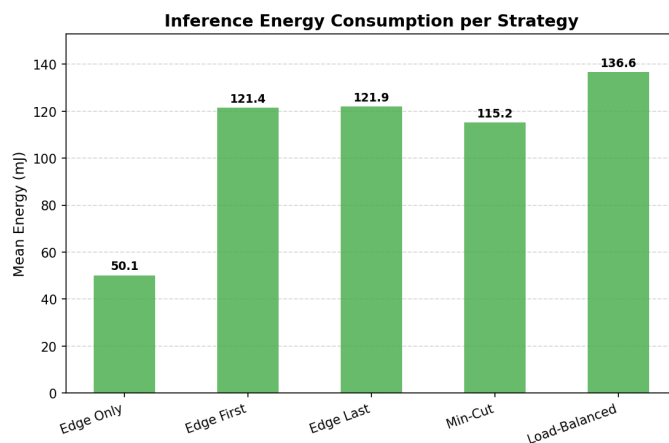


Figure 5.16: MLP Autoencoder energy consumption measurements.

Nonetheless, the single device inference originated from the Edge Only strategy is the most efficient in both metrics. For this model, the number of devices used in the strategies produces a direct proportional increase in latency. However, the energy consumption also increases, but not as significantly. The latter is due to the fact that the strategies shift from a mixed device approach, with the Edge First, Edge Last, and Min-Cut strategies, to a only **MCU** approach, with the Load-Balanced strategy, which is more energy efficient than the mixed device approach for a lighter computational load.

5.3 Overall Results Analysis

Considering the results from all the models, we can take note of a few patterns that emerge from using the different partitioning and distribution strategies.

Regarding latency, the Edge Only strategy consistently outperforms the other strategies, followed by the Edge First, Edge Last, and Min-Cut strategies, which have similar results between them. This pattern likely stems from the fact that these strategies rely on fewer devices, and therefore, fewer communication steps, which is a significant factor, as noticed in the active inference latency and overhead latency breakdown graphs.

Another pattern that emerges, especially in the **DS-CNN** model, is that both the Edge Only and Greedy strategies rely on a single device. Even so, the Greedy strategy takes longer to complete the inference process. The Argon **MCU** takes close to 100 times more time to communicate with the Orchestrator Model.

In terms of energy consumption, the results are shown to scale, firstly, with the number of devices that partake in the inference process, and secondly, with the type of devices that are used. The Edge Only strategies where only the edge device is used, are the most energy efficient approach, followed by the mixed device networks, and finally, the far-edge device only networks.

Even if not represented in the graphs, the idle energy consumption of the devices is a significant factor in the trade-off evaluation between the different strategies. The 10 times idle energy

consumption difference between the Argon MCU and the Raspberry Pi can be a determining factor, as the overall network idle time increases or decreases.

Nonetheless, each distribution strategy has its own merits and can be better suited to specific use cases, depending on the model architecture and network topology. The Edge First and Edge Last strategies are more suitable for networks where the MCUs devices are closer to sensors or actuators. The Min-Cut strategy highlights the importance of the intermediate tensor size, while the contrasting Greedy and Load-Balanced strategies emphasize the benefits of reducing the number of devices used for the inference process and, consequently, the number of communication steps.

In the end, the results indicate that the inclusion of edge devices in the inference process is only beneficial if the required throughput is above a certain threshold. This threshold depends on the context of the inference process, since the inclusion of edge devices only becomes beneficial when the inference load exceeds a given number of inference tasks per second. This is because edge devices have a higher power consumption, which is only justified when low latency is a requirement. The threshold can only be estimated for each specific use case, as it depends on the model architecture, the network topology, and the device characteristics, so no generalized threshold can be established. In other words, if the inference process is not time sensitive and time sparse, the inclusion of edge devices in the inference process is not beneficial.

Chapter 6

Conclusion

In the following sections, we will summarize the contributions of this dissertation and discuss the Research Questions that were addressed.

6.1 Contributions

Throughout this dissertation, we have presented **DisRUPT**, a model agnostic testbed that enables the evaluation of distributed inference for **MLPs** and **CNNs** models given a set of devices and a configuration file containing the model distribution, established routes and input data.

We have also highlighted a set of strategies and good practices for the deployment of **DNN** models in the edge environment, that lead to a lower inference latency and network energy consumption. Namely, the impact that communication steps have on the distributed inference process, and the importance of minimizing their occurrence, whether by reducing the number of partitions or by selecting certain partition points that minimize the amount of data that needs to be transmitted.

Taking into account the initial goal and research questions, as stated in Chapter 1, we can conclude that this work addressed all the research questions, as follows:

- **Hypothesis:** *"Is it possible to create a reliable tool to support the deployment of complex **DNN** models on far-edge and edge devices?"*

This hypothesis was confirmed, as the **DisRUPT** supports exactly that, even if limited in supported model architectures and width, by the underlying NAMP compatibility and the available resources of the devices.

- **RQ1:** *"Is it feasible to have collaborative ML inferences using edge and far-edge devices without compromising performance?"*

This research question was denied, although our work shows that the mentioned process is feasible, the performance is compromised in the sense that the inference latency is increased, even if not significantly.

- **RQ2:** *"What is the most efficient method to enable reliable cooperation between edge and far-edge devices partaking in an inference network?"*

During the experiments, we found out that, from the cooperative distribution strategies, there was no method that was generally more efficient than the others. And that efficiency is highly dependent on the model architecture, network conditions, and the available resources.

- **RQ3:** *"How efficiently can the resources of edge and far-edge devices be utilized towards collaborative inference tasks?"*

The experiments showed that the resources of edge and far-edge devices can be efficiently utilized, although the intermediate software layer that is required to enable inference on **MCUs** highly limits the available resources, and therefore, the efficiency of the process.

6.2 Future Work

Even though the contributions of this dissertation reached the expected goals, the work can be further extended in several ways that further evaluate the feasibility of having **DNN** execution in the edge environment and in which forms.

Firstly, the compatibility of the **DisRUPT** testbed can be expanded to support more **DNN** architectures, such as recurrent and attention layers, to enable the partitioning of more complex and capable models. Likewise, the support of intra-layer partitioning can be added to the testbed, removing the current limitation of only supporting inter-layer partitioning, and model width constraints.

Additionally, the testbed can be extended to support devices with accelerators, such as Tensor Processing Units (**TPUs**), Neural Processing Units (**NPU**s), and Graphics Processing Units (**GPU**s), allowing the evaluation of a more diverse range of strategies and deployment scenarios, to further explore the potential of distributed inference in the edge environment.

Finally, the orchestration and scheduling strategies can be further improved by implementing a real time computing scheduler that can dynamically adapt to the network conditions and the available resources. This results in quality improvements on the collected data, which will be closer to a real-world scenario, and therefore, more reliable and useful for future research in this field.

References

- [1] R. David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. arXiv:2010.08678 [cs, eess]. 2020. URL: <https://arxiv.org/abs/2010.08678>.
- [2] G. Al-Atat et al. “The Case for Hierarchical Deep Learning Inference at the Network Edge”. In: *Proceedings of the 1st International Workshop on Networked AI Systems*. NetAISys ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 13–18. ISBN: 979-8-4007-0212-9. DOI: [10.1145/3597062.3597278](https://doi.org/10.1145/3597062.3597278). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85164295939&doi=10.1145%2F3597062.3597278&partnerID=40&md5=683d8372200f23a97c4cf499ca414566>.
- [3] A.P. Behera et al. “Improved Decision Module Selection for Hierarchical Inference in Resource-Constrained Edge Devices”. In: *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. ACM MobiCom ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1501–1503. ISBN: 978-1-4503-9990-6. DOI: [10.1145/3570361.3615732](https://doi.org/10.1145/3570361.3615732). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85198962078&doi=10.1145%2F3570361.3615732&partnerID=40&md5=c75369a88f81dece6877d2790f9b9d65>.
- [4] Carsten Bormann and Paul Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. RFC Editor, Dec. 2020. DOI: [10.17487/RFC8949](https://doi.org/10.17487/RFC8949). URL: <https://www.rfc-editor.org/info/rfc8949>.
- [5] Mateusz Buda. *ALL-CNN*. <https://github.com/mateuszbuda/ALL-CNN>. 2017.
- [6] D. Chen et al. “Deep Reinforcement Learning for Collaborative Inference in Local- Edge Computing”. In: *Proceedings of the 2024 International Academic Conference on Edge Computing, Parallel and Distributed Computing*. ECPDC ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 138–142. ISBN: 979-8-4007-1816-8. DOI: [10.1145/3677404.3677428](https://doi.org/10.1145/3677404.3677428). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85202826259&doi=10.1145%2F3677404.3677428&partnerID=40&md5=185908bd6020c82533e4257a1884b0b5>.
- [7] E. M. K et al. “Adaptive Hierarchical Inference Framework for Enhanced Efficiency and Scalability in Complex IoT Decision-Making Systems”. In: *2025 IEEE 11th World Forum on Internet of Things (WF-IoT)*. Journal Abbreviation: 2025 IEEE 11th World Forum on Internet of Things (WF-IoT). Oct. 2025, pp. 1–3. ISBN: 2768-1734. DOI: [10.1109/WF-IoT64238.2025.11270664](https://doi.org/10.1109/WF-IoT64238.2025.11270664).
- [8] F. Sousa F. Rego J. Oliveira and L. Almeida. “NAMP: A Network-Aware Model Partitioning Framework for Constrained Devices”. In: *Proceedings of the 4th International Workshop on Real-time and Intelligent Edge computing (RAGE ’25)* (2025), pp. 1–6. DOI: [10.1145/3722567.3727844](https://doi.org/10.1145/3722567.3727844).

- [9] Mayank Gulati et al. “TDMiL: Tiny Distributed Machine Learning for Microcontroller-Based Interconnected Devices”. en. In: *IEEE Access* 12 (2024), pp. 167810–167826. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2024.3492921](https://doi.org/10.1109/ACCESS.2024.3492921). URL: <https://ieeexplore.ieee.org/document/10745482/> (visited on 12/29/2025).
- [10] Rakesh Halder, Krzysztof J. Fidkowski, and Kevin J. Maki. “Non-intrusive reduced-order modeling using convolutional autoencoders”. In: *International Journal for Numerical Methods in Engineering* (2022). DOI: [10.1002/nme.7072](https://doi.org/10.1002/nme.7072). URL: <https://doi.org/10.1002/nme.7072>.
- [11] Doyu Hiroshi, Roberto Morabito, and Martina Brachmann. “A TinyMLaaS Ecosystem for Machine Learning in IoT: Overview and Research Challenges”. In: (Apr. 2021), pp. 1–5. DOI: [10.1109/VLSIDAT52063.2021.9427352](https://doi.org/10.1109/VLSIDAT52063.2021.9427352).
- [12] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *arXiv preprint arXiv:1704.04861* (2017). URL: <https://arxiv.org/abs/1704.04861>.
- [13] Csaba Kiraly. *Anomaly Detection*. https://github.com/mlcommons/tiny/tree/master/benchmark/training/anomaly_detection. 2021.
- [14] B. Kitchenham et al. “Systematic Literature Reviews in Software Engineering - A Systematic Literature Review”. In: *Information & Software Technology* 51.1 (2009), pp. 7–15. DOI: [10.1016/J.INFSOF.2008.09.009](https://doi.org/10.1016/J.INFSOF.2008.09.009). URL: <https://doi.org/10.1016/J.INFSOF.2008.09.009>.
- [15] Y. Ma, Y. Wang, and B. Tang. “Joint Optimization of Model Partitioning and Resource Allocation for Multi-Exit DNNs in Edge-Device Collaboration”. In: *Electronics (Switzerland)* 14.8 (2025). DOI: [10.3390/electronics14081647](https://doi.org/10.3390/electronics14081647). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-105003660775&doi=10.3390%2Felectronics14081647&partnerID=40&md5=ed4bfff9eb6c599eafb4de99fdce789c6>.
- [16] Dat Tien Nguyen. *MobilenetV1*. <https://github.com/Nguyendat-bit/MobilenetV1>. 2021.
- [17] João Oliveira, Filipe Sousa, and Luís Almeida. “embServe: Embedded Services for Constrained Devices”. In: *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*. 2023, pp. 1–8. DOI: [10.1109/WFCS57264.2023.10144123](https://doi.org/10.1109/WFCS57264.2023.10144123).
- [18] P. Dai et al. “Joint Optimization of Device Placement and Model Partitioning for Cooperative DNN Inference in Heterogeneous Edge Computing”. In: *IEEE Transactions on Mobile Computing* 24.1 (Jan. 2025), pp. 210–226. ISSN: 1558-0660. DOI: [10.1109/TMC.2024.3457793](https://doi.org/10.1109/TMC.2024.3457793). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-86000377432&doi=10.1109%2FTMC.2024.3457793&partnerID=40&md5=71ec07013de9ebd073d2eee6fc0f90c5>.
- [19] R. G. Pacheco, R. S. Couto, and O. Simeone. “Calibration-Aided Edge Inference Offloading via Adaptive Model Partitioning of Deep Neural Networks”. In: *ICC 2021 - IEEE International Conference on Communications*. Journal Abbreviation: ICC 2021 - IEEE International Conference on Communications. June 2021, pp. 1–6. ISBN: 1938-1883. DOI: [10.1109/ICC42927.2021.9500760](https://doi.org/10.1109/ICC42927.2021.9500760). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85114267495&doi=10.1109%2FICC42927.2021.9500760&partnerID=40&md5=9c98cf9d174cc836fcf024d85a23ae29>.

- [20] Ramon Sanchez-Iborra and Antonio F. Skarmeta. “TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities”. In: *IEEE Circuits and Systems Magazine* 20.3 (2020), pp. 4–18. DOI: [10.1109/MCAS.2020.3005467](https://doi.org/10.1109/MCAS.2020.3005467).
- [21] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. RFC Editor, June 2014. DOI: [10.17487/RFC7252](https://doi.org/10.17487/RFC7252). URL: <https://www.rfc-editor.org/info/rfc7252>.
- [22] Peter Sørensen, Bastian Epp, and Tobias May. “A depthwise separable convolutional neural network for keyword spotting on an embedded system”. In: *EURASIP Journal on Audio, Speech, and Music Processing* 2020 (June 2020). DOI: [10.1186/s13636-020-00176-2](https://doi.org/10.1186/s13636-020-00176-2).
- [23] Jost Tobias Springenberg et al. “Striving for Simplicity: The All Convolutional Net”. In: *3rd International Conference on Learning Representations (ICLR), Workshop Track*. 2015. URL: <https://arxiv.org/abs/1412.6806>.
- [24] Bradley McDanel Surat Teerapittayannon and H. T. Kung. “Distributed Deep Neural Networks over the Cloud, the Edge and End Devices”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems* (2017), pp. 328–339. DOI: [10.1109/ICDCS.2017.226](https://doi.org/10.1109/ICDCS.2017.226).
- [25] T. Mohammed et al. “Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading”. In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. Journal Abbreviation: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. July 2020, pp. 854–863. ISBN: 2641-9874. DOI: [10.1109/INFOCOM41043.2020.9155237](https://doi.org/10.1109/INFOCOM41043.2020.9155237).
- [26] Nhu Y. Tran-Van and Kim Hung Le. “CoFANN: A collaborative framework for accelerating DNN inference in drone-based agricultural monitoring systems”. English. In: *Simulation Modelling Practice and Theory* 144 (2025). Type: Article. ISSN: 1569190X. DOI: [10.1016/j.simpat.2025.103176](https://doi.org/10.1016/j.simpat.2025.103176). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-105010432774&doi=10.1016%2Fj.simpat.2025.103176&partnerID=40&md5=ce7e1f3b22e087a80647e600be8468d4>.
- [27] W. Miao et al. “Adaptive DNN Partition in Edge Computing Environments”. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. Journal Abbreviation: 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS). Dec. 2020, pp. 685–690. ISBN: 2690-5965. DOI: [10.1109/ICPADS51040.2020.00097](https://doi.org/10.1109/ICPADS51040.2020.00097).
- [28] Hongli Wang et al. “Context-aware Adaptive Surgery: A Fast and Effective Framework for Adaptative Model Partition”. In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5.3 (Sept. 2021). Place: New York, NY, USA. DOI: [10.1145/3478073](https://doi.org/10.1145/3478073). URL: <https://doi.org/10.1145/3478073>.
- [29] Yundong Zhang et al. “Hello Edge: Keyword Spotting on Microcontrollers”. In: *arXiv preprint arXiv:1711.07128* (2018). URL: <https://arxiv.org/abs/1711.07128>.

Appendix A

Model Specifications

During the process of selecting and acquiring the models used in this dissertation, a series of characteristics were considered, including the model’s architecture, size, and layer-by-layer information. The following sections provide a detailed overview of the layer-by-layer details.

From Figure A.1 to Figure A.5, we present the layer-by-layer information for each model, including the layer type, input and output shapes, and output size. This information is crucial for understanding the feasibility of running the submodels on the microcontroller, as it provides insights into the computational requirements and memory usage of each layer.

Layer	Layer Type	Input Tensor Shape	Output Tensor Shape	Output Size
0 to 1	Conv2D	(None, 32, 32, 3)	(None, 32, 32, 12)	12.00 KB
1 to 2	Conv2D	(None, 32, 32, 12)	(None, 32, 32, 12)	12.00 KB
2 to 3	Conv2D	(None, 32, 32, 12)	(None, 16, 16, 12)	3.00 KB
3 to 4	Conv2D	(None, 16, 16, 12)	(None, 16, 16, 24)	6.00 KB
4 to 5	Conv2D	(None, 16, 16, 24)	(None, 16, 16, 24)	6.00 KB
5 to 6	Conv2D	(None, 16, 16, 24)	(None, 8, 8, 24)	1.50 KB
6 to 7	Conv2D	(None, 8, 8, 24)	(None, 6, 6, 24)	864 B
7 to 8	Conv2D	(None, 6, 6, 24)	(None, 6, 6, 24)	864 B
8 to 9	Conv2D	(None, 6, 6, 24)	(None, 6, 6, 10)	360 B
9 to 10	GlobalAveragePooling2D	(None, 6, 6, 10)	(None, 10)	10 B
10 to 11	Softmax	(None, 10)	(None, 10)	10 B

Figure A.1: All-CNN-C layer overview.

Layer	Layer Type	Input Tensor Shape	Output Tensor Shape	Output Size
0 to 1	Conv2D	(None, 49, 10, 1)	(None, 49, 10, 16)	7.66 KB
1 to 2	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
2 to 3	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
3 to 4	DepthwiseConv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
4 to 5	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
5 to 6	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
6 to 7	Conv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
7 to 8	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
8 to 9	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
9 to 10	DepthwiseConv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
10 to 11	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
11 to 12	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
12 to 13	Conv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
13 to 14	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
14 to 15	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
15 to 16	DepthwiseConv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
16 to 17	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
17 to 18	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
18 to 19	Conv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
19 to 20	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
20 to 21	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
21 to 22	DepthwiseConv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
22 to 23	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
23 to 24	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
24 to 25	Conv2D	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
25 to 26	BatchNormalization	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
26 to 27	ReLU	(None, 49, 10, 16)	(None, 49, 10, 16)	7.66 KB
27 to 28	GlobalAveragePooling2D	(None, 49, 10, 16)	(None, 16)	16 B
28 to 29	Dense	(None, 16)	(None, 12)	12 B

Figure A.2: DS-CNN layer overview.

Layer	Layer Type	Input Tensor Shape	Output Tensor Shape	Output Size
0 to 1	Conv2D	(None, 32, 32, 3)	(None, 16, 16, 8)	2.00 KB
1 to 2	BatchNormalization	(None, 16, 16, 8)	(None, 16, 16, 8)	2.00 KB
2 to 3	ReLU	(None, 16, 16, 8)	(None, 16, 16, 8)	2.00 KB
3 to 4	DepthwiseConv2D	(None, 16, 16, 8)	(None, 16, 16, 8)	2.00 KB
4 to 5	BatchNormalization	(None, 16, 16, 8)	(None, 16, 16, 8)	2.00 KB
5 to 6	ReLU	(None, 16, 16, 8)	(None, 16, 16, 8)	2.00 KB
6 to 7	Conv2D	(None, 16, 16, 8)	(None, 16, 16, 16)	4.00 KB
7 to 8	BatchNormalization	(None, 16, 16, 16)	(None, 16, 16, 16)	4.00 KB
8 to 9	ReLU	(None, 16, 16, 16)	(None, 16, 16, 16)	4.00 KB
9 to 10	DepthwiseConv2D	(None, 16, 16, 16)	(None, 8, 8, 16)	1.00 KB
10 to 11	BatchNormalization	(None, 8, 8, 16)	(None, 8, 8, 16)	1.00 KB
11 to 12	ReLU	(None, 8, 8, 16)	(None, 8, 8, 16)	1.00 KB
12 to 13	Conv2D	(None, 8, 8, 16)	(None, 8, 8, 32)	2.00 KB
13 to 14	BatchNormalization	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
14 to 15	ReLU	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
15 to 16	DepthwiseConv2D	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
16 to 17	BatchNormalization	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
17 to 18	ReLU	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
18 to 19	Conv2D	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
19 to 20	BatchNormalization	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
20 to 21	ReLU	(None, 8, 8, 32)	(None, 8, 8, 32)	2.00 KB
21 to 22	DepthwiseConv2D	(None, 8, 8, 32)	(None, 4, 4, 32)	512 B
22 to 23	BatchNormalization	(None, 4, 4, 32)	(None, 4, 4, 32)	512 B
23 to 24	ReLU	(None, 4, 4, 32)	(None, 4, 4, 32)	512 B
24 to 25	Conv2D	(None, 4, 4, 32)	(None, 4, 4, 64)	1.00 KB
25 to 26	BatchNormalization	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
26 to 27	ReLU	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
27 to 28	DepthwiseConv2D	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
28 to 29	BatchNormalization	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
29 to 30	ReLU	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
30 to 31	Conv2D	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
31 to 32	BatchNormalization	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
32 to 33	ReLU	(None, 4, 4, 64)	(None, 4, 4, 64)	1.00 KB
33 to 34	DepthwiseConv2D	(None, 4, 4, 64)	(None, 2, 2, 64)	256 B
34 to 35	BatchNormalization	(None, 2, 2, 64)	(None, 2, 2, 64)	256 B
35 to 36	ReLU	(None, 2, 2, 64)	(None, 2, 2, 64)	256 B
36 to 37	Conv2D	(None, 2, 2, 64)	(None, 2, 2, 128)	512 B
37 to 38	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
38 to 39	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
39 to 40	DepthwiseConv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
40 to 41	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
41 to 42	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
42 to 43	Conv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
43 to 44	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
44 to 45	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
45 to 46	DepthwiseConv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
46 to 47	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
47 to 48	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
48 to 49	Conv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B

Figure A.3: MobileNetV1 layer overview (part 1).

49 to 50	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
50 to 51	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
51 to 52	DepthwiseConv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
52 to 53	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
53 to 54	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
54 to 55	Conv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
55 to 56	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
56 to 57	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
57 to 58	DepthwiseConv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
58 to 59	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
59 to 60	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
60 to 61	Conv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
61 to 62	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
62 to 63	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
63 to 64	DepthwiseConv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
64 to 65	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
65 to 66	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
66 to 67	Conv2D	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
67 to 68	BatchNormalization	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
68 to 69	ReLU	(None, 2, 2, 128)	(None, 2, 2, 128)	512 B
69 to 70	GlobalAveragePooling2D	(None, 2, 2, 128)	(None, 128)	128 B
70 to 71	Dense	(None, 128)	(None, 256)	256 B

Figure A.4: MobileNetV1 layer overview (part 2).

Layer	Layer Type	Input Tensor Shape	Output Tensor Shape	Output Size
0 to 1	Dense	(None, 96)	(None, 96)	96 B
1 to 2	BatchNormalization	(None, 96)	(None, 96)	96 B
2 to 3	Activation	(None, 96)	(None, 96)	96 B
3 to 4	Dense	(None, 96)	(None, 96)	96 B
4 to 5	BatchNormalization	(None, 96)	(None, 96)	96 B
5 to 6	Activation	(None, 96)	(None, 96)	96 B
6 to 7	Dense	(None, 96)	(None, 96)	96 B
7 to 8	BatchNormalization	(None, 96)	(None, 96)	96 B
8 to 9	Activation	(None, 96)	(None, 96)	96 B
9 to 10	Dense	(None, 96)	(None, 96)	96 B
10 to 11	BatchNormalization	(None, 96)	(None, 96)	96 B
11 to 12	Activation	(None, 96)	(None, 96)	96 B
12 to 13	Dense	(None, 96)	(None, 8)	8 B
13 to 14	BatchNormalization	(None, 8)	(None, 8)	8 B
14 to 15	Activation	(None, 8)	(None, 8)	8 B
15 to 16	Dense	(None, 8)	(None, 96)	96 B
16 to 17	BatchNormalization	(None, 96)	(None, 96)	96 B
17 to 18	Activation	(None, 96)	(None, 96)	96 B
18 to 19	Dense	(None, 96)	(None, 96)	96 B
19 to 20	BatchNormalization	(None, 96)	(None, 96)	96 B
20 to 21	Activation	(None, 96)	(None, 96)	96 B
21 to 22	Dense	(None, 96)	(None, 96)	96 B
22 to 23	BatchNormalization	(None, 96)	(None, 96)	96 B
23 to 24	Activation	(None, 96)	(None, 96)	96 B
24 to 25	Dense	(None, 96)	(None, 96)	96 B
25 to 26	BatchNormalization	(None, 96)	(None, 96)	96 B
26 to 27	Activation	(None, 96)	(None, 96)	96 B
27 to 28	Dense	(None, 96)	(None, 96)	96 B

Figure A.5: MLP-Autoencoder layer overview.

Appendix B

Model Energy

To provide an estimate of the energy consumption of the models throughout the network, we have used a multimeter to measure the power consumption of each device during inference individually. Then, taking account for each submodel in each used strategy, we have estimated the energy consumption of each model by adding the corresponding segments. The following sections provide a detailed overview of the energy consumption of each model.

B.1 Submodel Energy Consumption Overview

In Table B.1, we present a comprehensive overview of the energy consumption of each submodel across the different models. The submodel identification is based on the beginning and ending layers of each submodel.

Model	Partition	Device	Energy Consumption (mJ)
All-CNN-C	0-8	Raspberry Pi	70,50
All-CNN-C	0-11	Raspberry Pi	56,60
All-CNN-C	3-7	Raspberry Pi	71,80
All-CNN-C	3-11	Raspberry Pi	67,90
All-CNN-C	0-3	Argon	88,90
All-CNN-C	3-6	Argon	72,30
All-CNN-C	6-9	Argon	40,60
All-CNN-C	6-11	Argon	33,90
All-CNN-C	7-11	Argon	27,20
All-CNN-C	8-11	Argon	24,00
All-CNN-C	9-11	Argon	31,50
DS-CNN	0-15	Raspberry Pi	70,70
DS-CNN	0-29	Raspberry Pi	54,10
DS-CNN	15-29	Raspberry Pi	68,30
DS-CNN	0-9	Argon	127,00
DS-CNN	0-29	Argon	26,50
DS-CNN	9-15	Argon	194,50
DS-CNN	15-21	Argon	194,60
DS-CNN	21-29	Argon	300,70
MobileNetV1	0-71	Raspberry Pi	71,00
MobileNetV1	36-71	Raspberry Pi	51,00
MobileNetV1	39-71	Raspberry Pi	54,10
MobileNetV1	0-15	Argon	91,40
MobileNetV1	0-24	Argon	69,60
MobileNetV1	15-27	Argon	58,80
MobileNetV1	24-36	Argon	25,10
MobileNetV1	27-39	Argon	37,40
MLP-Autoencoder	0-18	Raspberry Pi	49,60
MLP-Autoencoder	0-28	Raspberry Pi	50,10
MLP-Autoencoder	6-24	Raspberry Pi	49,60
MLP-Autoencoder	9-28	Raspberry Pi	51,20
MLP-Autoencoder	0-3	Argon	35,10
MLP-Autoencoder	0-6	Argon	30,90
MLP-Autoencoder	3-9	Argon	35,60
MLP-Autoencoder	6-18	Argon	33,90
MLP-Autoencoder	18-24	Argon	37,10
MLP-Autoencoder	24-28	Argon	34,70

Table B.1: Energy consumption overview by model, partition, and device