Hybrid Partitioning System for Embedded and Distributed CNNs Inference on Edge Devices.

Nihel Kaboubi

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Institut National de Recherche en Informatique et en Automatique

Système de partitionnement hybride pour une inférence distribuée et embarquée des CNNs sur les équipements en bordure de réseau.

Hybrid Partitioning System for Embedded and Distributed CNNs Inference on Edge Devices.

Présentée par :

Nihel KABOUBI

Direction de thèse :

Frédéric DESPREZ Directeur de recherche, INRIA Centre Grenoble-Rhône-Alpes	Directeur de thèse
Thierry COUPAYE Docteur en sciences HDR, Orange SA	Co-directeur de thèse
Loïc LETONDEUR Ingénieur Docteur, Orange SA	Co-encadrant de thèse

Rapporteurs :

Adrien LÈBRE
PROFESSEUR DES UNIVERSITES, IMT Atlantique
Sara BOUCHENAK
PROFESSEUR DES UNIVERSITES, INSA Lyon

Thèse soutenue publiquement le **25 avril 2023**, devant le jury composé de :

Adrien LÈBRE PROFESSEUR DES UNIVERSITES, IMT Atlantique	Rapporteur
Sara BOUCHENAK PROFESSEUR DES UNIVERSITES, INSA de Lyon	Rapporteure
Jean-Marc NICOD PROFESSEUR DES UNIVERSITES, École Nationale Supérieure de Mécanique et des Microtechniques (ENSMM)	Président du jury
Vania MARANGOZOVA MAITRE DE CONFERENCES, Université Grenoble Alpes	Examinatrice
Eddy CARON MAITRE DE CONFERENCES, ENS Lyon	Examinateur

Invités :

Frédéric DESPREZ
DIRECTEUR DE RECHERCHE, Centre Inria de l'Université Grenoble Alpes
Thierry COUPAYE
INGENIEUR HDR, Orange SA
Loïc LETONDEUR
INGENIEUR DOCTEUR, Orange SA
Denis TRYSTRAM
PROFESSEUR, Grenoble INP

Acknowledgement

Here I am, at the end of my Ph.D. This long and challenging journey full of adventures and surprises. This journey would not have been possible without the support, guidance, and feedbacks from numerous people around me.

I would like to express my sincere gratitude to my thesis supervisor, Loïc Letondeur, for his guidance, support, and patience throughout my research journey. His insightful feedback and constructive criticism have helped me shape my ideas and refine my arguments. I am grateful for his unwavering encouragement.
I would like to express my deepest appreciation to my Co-supervisor Thierry Coupaye for his advice, guidance, and valuable suggestions that helped me at various stages of my research. I am also grateful to Frédéric Desprez, and Denis Trystram for their consistent support and guidance during the running of this project.

I would like to thank Adrien Lébre and Sara Bouchenak for accepting to review this thesis as well as for their valuable insights. I would also like to express my sincere gratitude to the Jury members Jean-Marc Nicod, Vania Marangozova and Eddy Caron for their challenging questions and appreciations during my thesis defense.

I would also like to thank my family for their unwavering encouragement and support throughout my academic journey, especially my father Faouzi kaboubi and my husband Majdi Bali, who have been my pillars of strength and support. Their love and support have been a constant source of motivation for me.

Additionally, I would like to thank my work team in Orange, my colleagues and friends for their support and encouragement. Their intellectual contributions and stimulating discussions have enriched my research and made this journey more enjoyable.

I am grateful to the participants who generously gave their time and insights to make this study possible. Without their contributions, this research would not have been possible.

As the famous motivational song goes:

"If you believe, you can move the highest mountains
Cross the greatest oceans, walk across the water, the water
Believe, you can move the highest mountains
Cross the greatest oceans, walk across the water
If you believe "

If you believe
Strive to Be & Patch Crowe

Thank you to everyone who has been a part of my academic journey and helped me reach this milestone. Your support and encouragement mean the world to me, and I am forever grateful.

Abstract

Title : Hybrid Partitioning System for Embedded and Distributed CNNs Inference on Edge Devices

The combination of Edge Computing and Artificial Intelligence (AI) technologies offers additional perspectives for creating innovative and efficient applications for the Internet of Things. Convolutional Neural Networks (CNNs) and Deep Neural Networks are used in many applications, especially for real-time analysis for data such as images, sounds or videos. To address some issues such as resilience, confidentiality and responsiveness, often voluminous models must then be inferred on devices at the Edge.

However, even in inference, these models consume significant memory and computational resources that can exceed the capabilities of most Edge devices. Also, these edge devices are subject to breakdowns and/or hardware defects, which negatively impacts the ability to constitute a reliable infrastructure. State of the art shows that the current solutions to adapt AI models inference to the Edge are insufficient to address these issues. Existing solutions may require a new phase of training, transmission of data to the cloud and/or lead to a potentially significant degradation in accuracy.

In order to cope with these problems, the work presented in this manuscript proposes HyPS (Hybrid Partitioning System), a solution based on a hybrid partitioning strategy. This strategy aims to identify the best positions to divide the CNN structure into a set of partitions whose size is determined to consider the constrained resources of the Edge devices. HyPS distributes, organizes and makes a reliable CNNs inference on resources-constrained devices. The proposed approach has been validated experimentally thanks to a prototype allowing a proof of concept of both the applied strategy and the architecture proposed for HyPS. Several use cases show the interest of HyPS for distributed, embedded, and reliable CNNs inference at the Edge.

Keywords: edge computing , model partitioning, distributed inference, internet of things, AI technologies

Résumé

Titre : Système de partitionnement hybride pour une inférence distribuée et embarquée des CNNs sur les équipements en bordure de réseau.

La combinaison de l'informatique en bordure de réseau (Edge Computing) et des techniques d'Intelligence Artificielle (IA) offre des perspectives supplémentaires pour créer des applications innovantes et efficaces pour l'Internet des objets. Les réseaux de neurones convolutifs (CNNs) et les réseaux de neurones profonds sont utilisés dans de nombreuses applications, en particulier pour l'analyse en temps quasi-réel pour des données telles que des images, des sons ou des vidéos. Pour adresser des problématiques de résilience, de confidentialité et de réactivité, des modèles souvent volumineux, doivent alors être déployés en inférence sur les devices présents dans le Edge.

Cependant, même en ce qui concerne uniquement la phase d'inférence, ces modèles consomment des ressources de mémoire et de calcul importantes qui peuvent être supérieures aux capacités de la plupart des devices du Edge. D'autre part, ces mêmes devices sont sujets à des pannes et/ou des défauts matériels ce qui impacte négativement leur capacité à constituer une infrastructure digne de confiance. L'état de l'art montre que les solutions actuelles pour adapter l'inférence de modèles conséquents d'IA au Edge sont insuffisantes. Les solutions existantes peuvent effectivement nécessiter une nouvelle phase d'entrainement, la transmission de données vers le cloud et/ou entraîner une dégradation potentiellement significative de la précision.

Pour faire face à ces problèmes, les travaux présentés dans ce manuscrit proposent HyPS (Hybrid Partitioning System), une solution bâtie autour d'une stratégie de partitionnement hybride. Cette stratégie vise à identifier les meilleurs emplacements pour diviser la structure d'un CNN en un ensemble de partitions dont la taille est déterminée pour tenir compte des ressources contraintes des devices du Edge. HyPS permet de répartir, organiser et fiabiliser l'inférence de CNNs sur des devices dont les ressources sont trop contraintes. L'approche proposée a été validée expérimentalement grâce à un prototype permettant une preuve de concept de la stratégie utilisée ainsi que de l'architecture proposée pour HyPS. Plusieurs cas d'usage permettent de montrer l'intérêt de HyPS pour une inférence distribuée, embarquée et fiabilisée de CNNs dans le Edge.

Mots-clés: traitement des données à la périphérie du réseau, partitionnement de modèle, inférence distribuée, internet des objets, technologies d'intelligence artificielle

Contents

List of Figures

List of Tables

Publications

International conference paper

Nihel Kaboubi, Loïc Letondeur, Thierry Coupaye, Fréderic Desprez and Denis Trystram. "Hybrid Partitioning for Embedded and Distributed CNNs Inference on Edge Devices." 2022, ANTIC International conference on Advanced Network Technologies and Intelligent Computing, 2022.

A patent application is filed on October 14, 2022 and registered under the reference FR2210604 titled "Procédé de distribution des paramètres d'un réseau de neurones, un procédé d'inférence et les dispositifs associés".

List of abbreviations

The following list describes the significance of abbreviations used throughout this thesis. This list is made per chapter to show where each abbreviation appears first. No new entry will be made if a certain abbreviation returns in a later chapter.

AI	Artificial Intelligence
IoT	Internet of Things
DNN	Deep Neural Network
CNN	Convolutional Neural Network
POP	Point Of Presence
MEC	Multi-access Edge Computing
KD	Knowledge distillation
MEC	Mobile Edge Computing
DAG	Directed Acyclic Graph
IONN	Incremental Offloading of Neural Network
ML	Machine learning
FCFS	first come, first served
VM	Virtual Machine
MCC	Mobile Cloud Computing
NN	Neural Network
ANN	Artificial Neural Network
MDP	Markov Decision Process
DRL	Deep Reinforcement Learning
DINA	Distributed INference Acceleration
QoS	Quality of Service
FL	Federated Learning
QAT	Quantization-Aware Training
PQT	Post-training quantization
DAG	Directed Acyclic Graph
DINA	Distributed INference Acceleration
UAV	Unmanned Aerial Vehicle
HMTD	Hierarchical Machine learning Tasks Distribution
MAC	multiply–accumulate
HyPS	Hybrid Partitioning System
SoC	System On a Chip
MQTT	Queuing Telemetry Transport
AMQP	Advanced Message Queuing Protocol
QoS	Quality of Service
GAN	Generative neural network
LSTM	Long Short-Term Memory
DFA	Deterministic Finite Automata

Chapter 1

Introduction

1.1 Context and Problem Statement

The convergence of the Internet of Things (IoT) and Artificial Intelligence (AI) led to a highly automated future and a highly automated cyber-physical world. IoT is an emerging paradigm that enables communication and data exchange between connected devices and sensors through the internet [1] [2]. In 2022, the market for the IoT is expected to grow 18% to 14.4 billion active connections. It is expected that by 2025, there will be approximately 27 billion connected IoT devices [3]. Connected devices will be deployed in homes, buildings, vehicles, cities, and industries.

AI enabled IoT creates intelligent machines that simulate smart behavior and supports in decision making. AI makes the devices learn from their data and interact with the surrounding environment thanks to Deep Learning(DL) technologies such Deep Neural Networks(DNNs) and noticeably Convolutional Neural Networks(CNNs).

IoT does not operate alone as DL requires computation storage and communication facilities to provide users for advanced IoT applications. These facilities are typically located today into the Cloud. For example, CNNs are used in a wide variety of tasks such as image recognition, video analysis, and object detection [4] [5]. They are typically deployed on remote cloud servers and users data are generally collected and then uploaded to remote cloud servers for training and inference. This constant and silent upload results in a strong usage of the network and transmission of massive data which is an important drawback regarding environmental concerns.

Another drawback while working with cloud computing services concerns privacy [6][7]. Cloud storage can be easily accessible over the internet. Consequently, data can be accessed from anywhere on the internet in case of a data breach through hacking [8] [9]. Thus, cloud technology is not entirely secure, and users data could be compromised. Additionally, cloud computing introduces network congestion and latency that can devalue service delivery and causes significant bandwidth costs. The significant end-to-end delay from end-devices to the Cloud servers that are often too far from end-users can deter the performance of applications that require real-time analysis, such as instant messaging (IM) applications, online gaming, video applications, etc [10].

The alternative solution is to relocate the AI models at network border and optimize performance by avoiding round trips to the cloud or other centralized information systems. Edge computing [11] [12] paradigm emerges as an extension of cloud computing to move computing resources from clouds and data centers to the proximity of end users [12]. It unfolds cloud computing by extending the computation, storage, and resources to the edge of the network, close to the data source, to accomplish critical needs of real-time servicing, application intelligence, security, and privacy [13]. Edge computing provides intelligent, responsive and fault-tolerant IoT services. Figure 1.1 shows an overview of the edge computing infrastructure. Edge computing has several advantages but still suffers from limitations related to limited amount of data. Edge Computing technologies have limitations regarding memory, the ability to store a very large amount of data is also limited [14] [15].

The edge computing paradigm comprises thousands of IoT devices in a continuum rang-

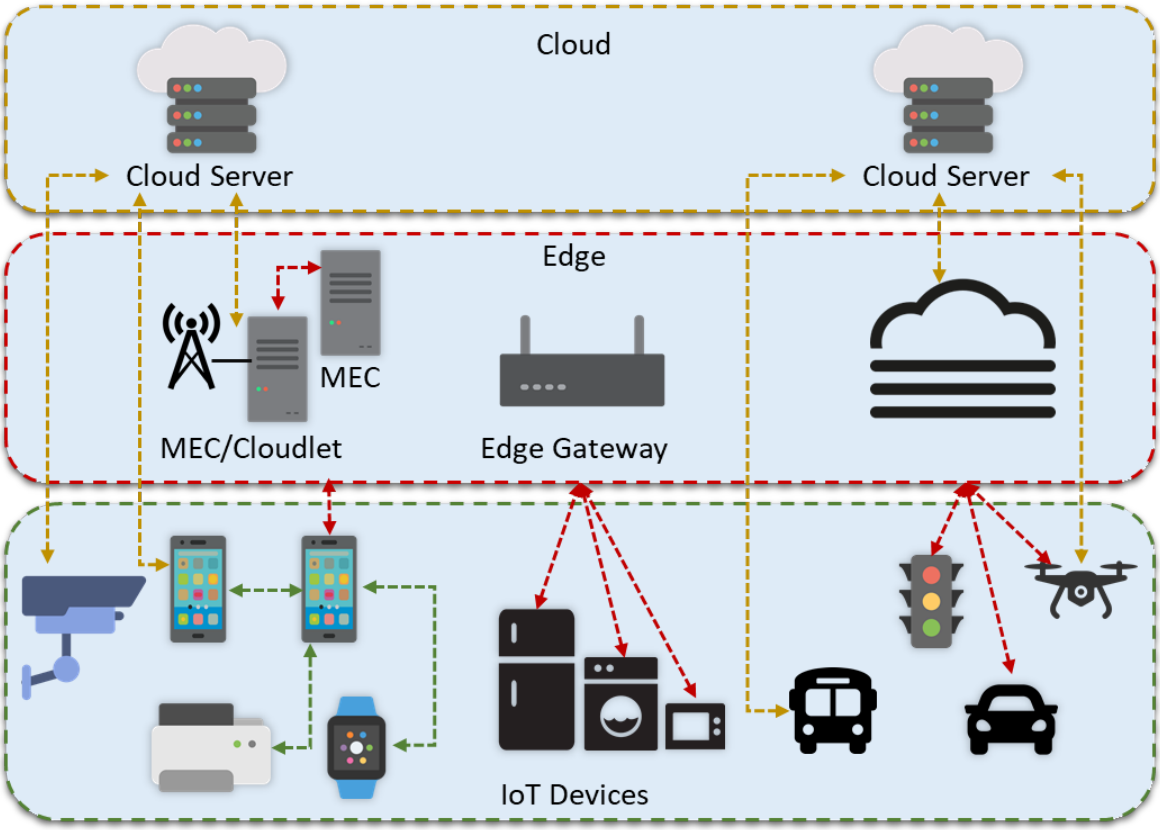

Figure 1.1: Edge computing infrastructure.

ing from edge devices to cloud servers. Such edge devices like sensors, routers, and Raspberry Pis, many with limited resources, will gather data and interact with the surrounding environment. Some of which may spend most of the time executing little work while others will need to execute heavy applications using intelligent tasks based on AI models.

This process is known as *edge intelligence*, also known as Edge AI, i.e., Edge computing applied to AI [16]. Edge devices must leverage DL models to implement accurate predictions, make decisions, and decode behavior behind sensors' data. Figure 1.2 illustrates the benefits of the edge computing concept.

Nevertheless, these pre-trained models often present a high computational cost, which brings more challenges in using resource-limited devices even if it is only considered for executing the inference phase of these models. Performing inference in the cloud can be a problem for some critical applications for big companies like Orange, which nevertheless has an extensive infrastructure at the network periphery (examples: relay antennas, network point of presence (POPs), internet access boxes, etc.).

Furthermore, inferring pre-trained large DNNs consumes significant time, memory, and computational resources that can be incompatible with most of the edge devices capabilities. Apart from hardware capabilities, edge devices often suffer from failures and corruption that result in generating erroneous data and causing unpredictable service loss [17]. DNN execution on edge devices needs to be robust enough to cope with a satisfactory quality of service from the user point-of-view. Because edge devices are often located in unprotected

Figure 1.2: Benefits of edge computing.

areas (e.g. customers' home), another drawback of edge intelligence is related to secrets protection. This topic is tremendous owing to the risks of disclosures concerning data and the DNN models themselves.

Training a large AI model on an edge infrastructure is very costly in terms of energy and time. Re-training a DNN model involves also financial costs and negative impact on environment. A study by researchers at the University of Massachusetts [18] shows that the process of training AI models can emit more than 626,000 pounds of carbon dioxide(CO_2) equivalent—nearly five times the lifetime emissions of the average American car (and that includes manufacture of the car itself). Training advanced AI models require high-powered GPU to run. In addition, this training on power-intensive GPUs contributed to increase CO_2 emissions which is very harmful for the environment.

Therefore, it is more efficient to capitalize on existing training without requiring any re-training phase. Hence, taking advantage of pre-trained models can reduce the carbon footprint significantly.

In the following section, we highlights the main contributions of this thesis.

1.2 Contributions

The thesis work presented in this manuscript aims to perform large AI models, particularly CNN model inference, on edge infrastructure. Some existing approaches intend to fill the gap between the resource demands of CNNs models and edge devices' capabilities for inference, but still, many issues need to be resolved. Partitioning a large CNN structure into small partitions is one of the most popular existing approaches adopted to run AI models at the edge. Running partitioned CNN inference at the edge led to many advantages listed as follows:

- **Avoids network congestion and bandwidth saturation:** model partitions are deployed at the edge ,and the collected data is processed locally without any transmission in the network.
- **Improves resiliency and reliability:** partitions can be distributed across multiple independent devices. One partition can be deployed on different devices which allowing opportunities for a recovery in the case of breakdown. Moreover, if there is a problem at one edge device, the inference process can be resumed and continued with the other devices.
- **Enhances data and model protection:** edge devices still exposed to hacking vulnerability. However, each device has only a part of the global model and data which is hardly a complete collection of data that hackers can pounce on. Privacy can easily be compromised when data hosted on centralized servers are hacked because they contain more significant data about users.

Therefore, this work proposed a Hybrid partitioning strategy that allows CNN partitioning and inference without impacting accuracy, without requiring to re-train a model, and without complex computations before deployment.

The main contributions of this work are listed below:

- **A Hybrid Partitioning System called HyPS** to make effective partitioning of a large CNN model by identifying the best partitioning strategies while minimizing the communication overhead and inference response latency. **Partitioned CNNs can be easily executed on one device or can be distributed across a cluster of multiple edge devices. HyPS does not modify the original NN structure and do not require any retraining step. HyPS keeps the exact accuracy of the original model**,
- **An orchestration architecture** for distributed inference of partitioned DNN which exhibits good properties in terms of reliability, resilience and privacy,
- **A prototype** that implements the functional behaviour of the proposed concepts of HyPS used to set up the testbed,
- **a scheduling policy** to establish a coordinated execution of the model partitions and ensure data tracking.
- **An evaluation** of the proposed hybrid partitioning strategy compared to existing approaches and analysis of the experimental results.

1.3 Thesis Structure

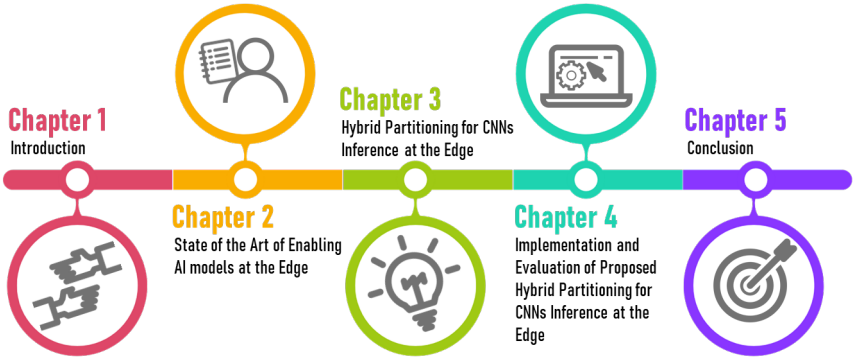

Figure 1.3: Thesis Outline.

Apart from the introductory Chapter 1, this thesis consists of four chapters organised as illustrated in Figure 1.3 :

Chapter 2: State of the Art of Enabling AI models at the edge This chapter reviews existing relevant works that allow AI models at the edge. Some researchers adapt DNN models to fit edge devices by reducing the model's size using model compression techniques. Others partition the model into partitions and perform distributed inference across edge devices. We mainly focus on existing partitioning strategies that allow large DNNs to fit resource-constrained devices. This chapter compares the most used partitioning strategies and highlights the strength and weaknesses of each approach.

Chapter 3: Hybrid Partitioning for CNNs Inference at the Edge

This chapter details the hybrid partitioning strategy, the architecture design for the distributed inference process, and the scheduling policy adopted to control the model partition execution. We define vertical and horizontal partitioning strategies and the specificity of our proposed strategy. HyPS is explained in details based one example of CNN that is VGG16. This chapter involves a prototype and an architecture design to be used in the implementation and evaluation of HyPS. In addition, qualitative validation of the proposed architecture is realized via citation of real-world use cases in which applying HyPS will bring several improvements.

Chapter 4: Implementation and Evaluation of Proposed Hybrid Partitioning for CNNs Inference at the Edge

This chapter includes two parts. Part one presents the experimental setup and the implementation details. The second part contains several experiments to evaluate the hybrid partitioning strategy applied to VGG16. Through these experiments, we analyze the impact of hybrid partitioning on communication overhead and inference time. We perform partitioned VGG16 inference on a single device and distribute it across multiple devices. This chapter shows the efficiency of HyPS on a real test bed.

Chapter 5: Conclusion

The closing chapter of the thesis provides a restatement of our work and discussion of some challenges. Furthermore, a detailed review of our contributions of our approach are presented. The final section we outline technical and research perspectives for future work.

Chapter 2

State of the Art of Enabling AI models at the edge

2.1 Introduction

As described in the introduction chapter, this thesis addresses the problem of performing AI applications that include large DNNs at the edge. Chapter 1 outlines the main motivations and issues underpinning the Edge Intelligence paradigm's importance. In this chapter a review of the main existing works to deploy DNNs on edge devices has been pursued. These concepts are essential for comprehending the positioning and the interest in the work provided in this thesis. The section 2.2 provides some background namely basic concepts about CNN structure and CNNs partitioning strategies.

Two types of existing solutions adapt DNNs to be deployed on edge to address this issue. First, by reducing the model size using specific algorithms that modify the model architecture and make it lighter. Second, partitioning a DNN structure into small partitions that can be distributed and deployed separately on IoT devices. The research studies are separated into two main sub-sections for clarity of exposition. First, section 2.3 discusses popular techniques for adapting DNN models to edge infrastructure. Second, the section 2.4 contains important approaches suggested to partition and distribute DNNs computations across edge devices.

2.2 Background

2.2.1 Overview of Convolutional Neural Networks Structure

A CNN is, in most cases, a DNN model specialized in processing data with a grid-like topology, such as an image. Their applications include image and video recognition, classification, computer vision, sound recognition through spectrogram analysis and natural language processing. CNN architecture is essentially composed of two parts: *feature extractor* and *classifier*. Each part is made of several layers. Feature extractor layers process the original input, and classifier layers then classify the resultant features. The inference delay of these two parts differs from one model to another and depends on CNNs structures. The particularity in CNN structure is that the neurons in the CNN layers are comprised of neurons organised into three dimensions, the spatial dimensionality of the input (height and the width) and the depth.

A CNN model mainly includes three types of layers, *convolution layers (Conv), pooling layers (Pool), batch normalization layers (BN)*, and *fully connected layers (FC)*. Each layer computes and generates a feature map as output that will be then the input of the next layer.

- **Convolution layers:** their purpose is to extract features in the images received as input. This is done by using the convolution filter named *Kernel*. The kernel is a matrix, which is slid across the image and multiplied with the input image. This procedure is repeated by applying multiple kernels to form an arbitrary number of feature maps, which represent different characteristics of the input data[19] [20],

- **Pooling layers:** this type of layer is often placed between two layers of convolution. The pooling layers receive multiple feature maps and apply a down-sampling operation. The pooling operation consists in reducing the size and dimensionality of the images while preserving their main characteristics. the different types of pooling operations are: maximum Pool, minimum Pool, average Pool, and Adaptive Pool,
- **Fully-connected layers (dense layers):** refers to a Multi layer perceptron. Multiple dense layers are connected in such a way that the outputs of one layer are fully connected to the inputs of the next layer. This type of layer applies a linear combination and possibly an activation function to the input values. Finally, fully-connected layer calculates final probabilities for each class which is the final CNN output.

A CNN is generally structured in a pipeline. Therefore, it can be deployed at the Edge according to two modes methods: adapting the CNN structure or partitioning the CNN model into small partitions. Each layer only depends on the previous layer's output and not on the other layers.

2.2.2 Terminology : CNN Model Partitioning Strategies

The following terms are consistently employed throughout this thesis so as to avoid confusion. This paragraph contains the coherent definitions and details. In this manuscript, *partitioning* refers to the splitting of an entire model onto specific points of its architecture to obtain one or more partitions. V-partition is the resultant of a vertical partitioning. H-partition is the resultant of a horizontal partitioning

To better explain the different partitioning strategies, we consider a CNN model whose each layer is denoted L_i that will be partitioned using three partitioning strategies. For the sake of comprehension, the CNN structure will be partitioned into four partitions denoted P_i, but other partitioning possibilities exist. According to the strategy applied the content of partitions changes as depicted in fig 2.1, 2.2 and 2.3.

Vertical model partitioning consists in building partitions made of one or several complete layers named V-partition. In Figure 2.1, each partition includes a set of consecutive layers. Each layer keeps its weights and parameters already fixed before inference. The input data of the intermediate partition is the output data generated by the previous partition. Only the first partition received the original input data.

Horizontal model partitioning which partitions the weights across layers. In Figure 2.2 all the layers are splitted horizontally. A partition can contain one or more units from different layers. The input data are sent to all partitions. Different partitions of the same splitted layers must communicate and synchronize with each other because all the output data must be concatenated with the output data from the other partitions to get the final output.

Data partitioning consists in dividing the original data given as input to all the CNN partitions. Weight partitioning is not considered, and each partition includes all layers of the CNN model. For data partitioning partitions P_i include exactly the same layers but

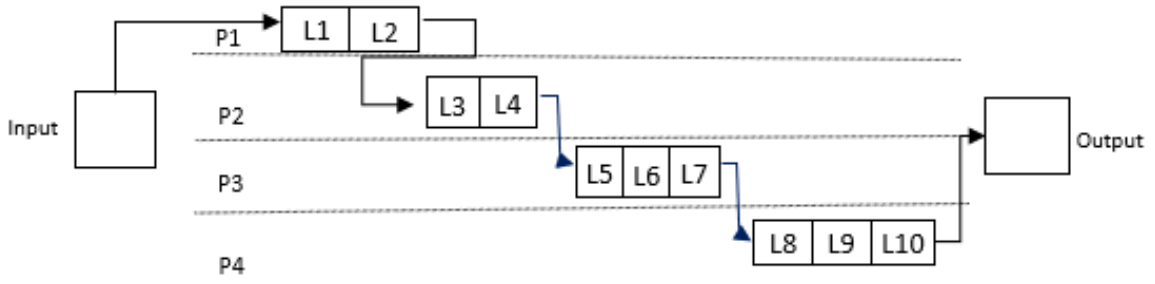

Figure 2.1: Vertical partitioning of 10 layers (L1..L10) into 4 V-Partitions (P1..P4).

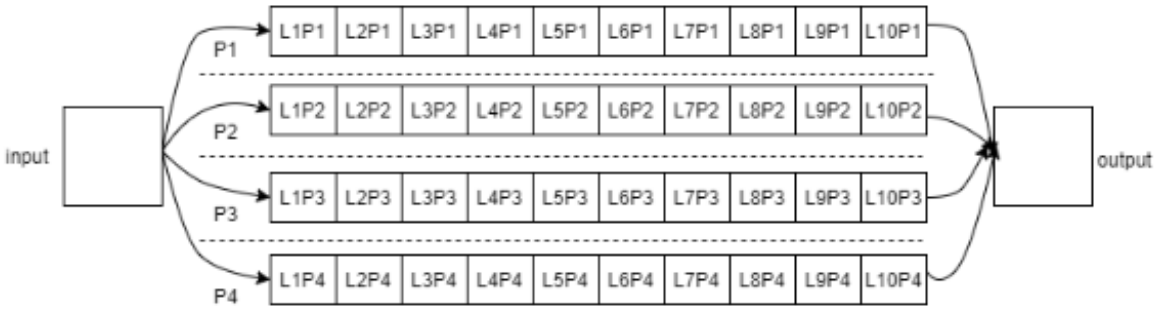

Figure 2.2: Horizontal partitioning. Source[21]

Figure 2.3: Data partitioning.Source[21]

receive only a part of the original data. After the computation of all partitions, outputs may be fused to get the final inference output.

For the better understanding and evaluation of the existing approaches, we have defined a grid that includes the most important criteria that interest us in each approach result. The criteria grid is as follows:

- **Model structure changes**: specifies if the proposed approach modifies the original model structure or not

- **Accuracy loss:** specifies if the proposed approach decreases the model accuracy or not ;
- **Model re-training requirement:** precises if the proposed approach requires model re-training or not ;
- **Partitioning type:** specifies the partitioning strategy used to split the original model;
- **Partitions placement** specifies where the partitions are deployed;
- **Running on single edge device** specifies if the approach allows to run large DNN model on single resource-constrained device;
- **Scheduling policy:** specifies if the approach adopt a scheduling policy for the model inference or not.

Figure 2.4 introduces a 'taxonomy' of the existing works studied and discussed in Sections 2.3 and 2.4.

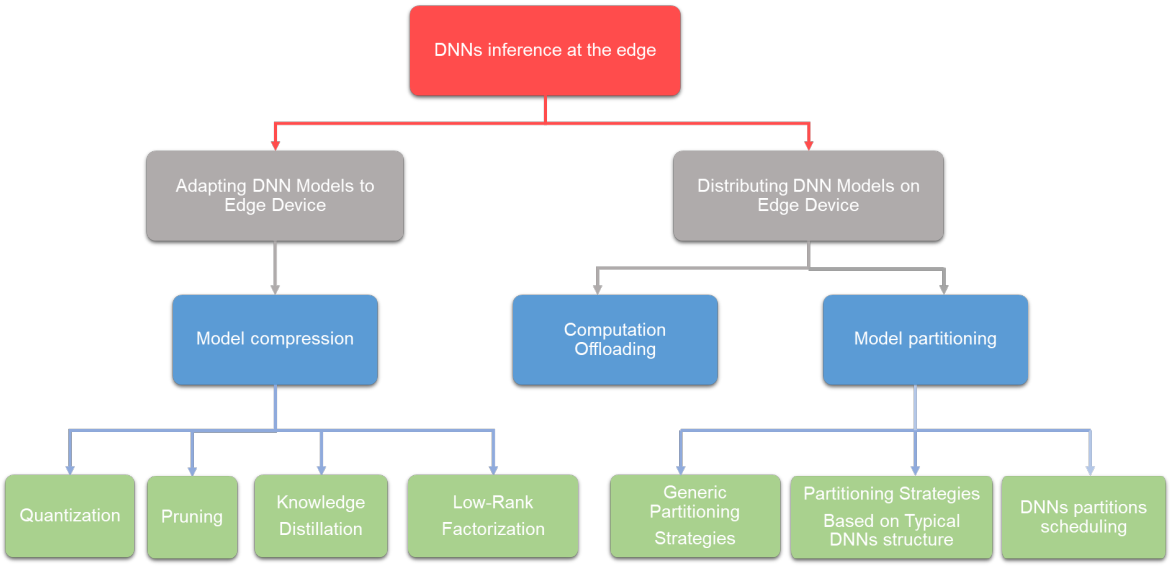

Figure 2.4: Mindmap of existing approaches performing DNNs inference at the edge

2.3 Model Compression: Adapting DNN Models to Edge Devices

Achieving efficient, real-time NN inference with optimal accuracy requires rethinking NN models' design, training, and deployment. Amount of research has focused on addressing these issues by making DNNs models more efficient (in terms of latency, memory footprint,

energy consumption, etc.), while still providing near optimal accuracy/generalization trade-offs thanks to *compression*. Due to the massive computational requirements of recent DNN models and the ubiquitousness of edge devices, the need for model optimization techniques is gaining in popularity. DNN models requires powerful machines with high computation and storage capacities. Numerous studies discuss different types of methods for model compression of such DNN models to enable their deployment at the edge.

Edge devices are often limited by memory, and computation constraint that motivate the development of compressed DNN models. Model compression is one of the most effective techniques to deploy DNN models on edge devices efficiently. Model Compression broadly reduces model size and latency overhead. *Size reduction* focuses on making the model lighter by reducing model parameters, thereby reducing RAM requirements in execution and storage requirements. *Latency reduction* refers to decreasing the time a model takes to make a prediction or infer a result. Model size and latency often go together, and most techniques reduce both. There are four main categories of model compression techniques: *quantization, pruning, Knowledge distillation(KD)*, and *low-rank factorization*.

2.3.1 Quantization

Model quantization compresses the number of bits used to encode each weight of the original neural network, so that the total memory footprint is reduced by the same factor[22]. Quantization can be performed using integer rather than floating-point data types. This is something interesting as integer operations require much less computations. The weights can be quantized to 16-bits, 8-bits, 4-bits, or even with 1-bit (which is a particular case of quantization in which weights are represented with binary values only, known as weight binarization). There are several types of quantization. Going from *float32* to *int8* or quantizing from *float32* to *float16*. Quantization can significantly reduce the number of Multiply-and-Accumulate (MAC) operations of a DNN model. Quantization has a significant impact on DNNs. If we consider float32 parameters quantized to float16, the occupied memory will be reduced [23] and probably no accuracy loss [24], but the speed will not increase. On the other hand, quantizing with *int8* can result in much faster inference, but the performance will probably be worse [25]. It won't even work in extreme scenarios and may require quantization-aware training.

In practice, there are two principal ways to do quantization. Quantization-aware training (QAT) quantizes the weights during training. However, the main drawback of QAT is the computational cost of re-training the NN model. This re-training may need to be performed for several hundred epochs to recover accuracy, especially for low-bit precision quantization [26]. Second, Post-training quantization(PQT) consists of quantizing model weights after training. Its main advantage is its simplicity of applying when limited or unlabeled data. However, it causes performance degradation and accuracy loss.

In 1948, Shannon wrote his seminal paper on the mathematical theory of communication [27], which formally presented the effect of quantization and its use in coding theory. He argued that a more optimal approach would be to vary the number of bits based on an event's probability, a concept now known as variable-rate quantization. In [28], authors

demonstrate the feasibility of deploying CNN models with post-training quantization to detect benign and malignant breast cancer tumors on portable ultrasound devices. This work shows that the size of CNN models is significantly reduced after applying quantization techniques. However, quantization generally yields drastically degraded accuracy.

Gong et al.[29] applied a quantization technique to optimize CNN models. The vector quantization methods reduced the storage requirements of CNNs. Results of this work show that the original parameters could be compressed by up to 96 % while retaining 99 % of the original accuracy measure. Zhou et al.[30] presented incremental network quantization (INQ). This method replaces all weights with powers of two or zero, iteratively, in each iteration, preserving some weights in full precision and retraining them. After multiple iterations, most weights are converted to a power of two. INQ allows efficient and low-power inference: they require considerably less memory and have lower computational complexity since quantized values can be stored, multiplied, and accumulated efficiently. However, this quantization method causes accuracy loss and performance degradation.

In another work, authors in [31] introduce the first practical 4-bit post-training quantization approach: it does not involve training the quantized model (fine-tuning). They adopt knowledge about the statistical characterization of neural network distributions to design efficient quantization schemes that minimize the mean-squared quantization error at the tensor level, avoiding retraining. In the same direction, Yoni et al. in [32] address the quantization problem for weights and/or activations of a pre-trained NN on highly constrained hardware, wherein complete retraining or mixed precision calculations cannot be tolerated. This work provides empirical evidence that input quantization is responsible for a significant part of the accuracy loss, notably on low-bit representation.

Jacob et al. [33] proposed a quantized inference framework that quantizes both weights and activations as 8-bit integers and just a few parameters (bias vectors) as 32-bit integers. This framework allows inference to be carried out using integer-only arithmetic, which can be implemented more efficiently than floating-point inference on commonly available integer-only hardware. Previous research works Han et al.[34] takes a pre-trained model and passes it through the three-stage pipeline that consists of pruning the weights, quantization of the weights, and finally, Huffman encoding them for a 35-49x reduction in model size. During this process, they re-train the network multiple times to ensure almost no accuracy is lost.

Post-training quantization (PTQ) is supported and preferable to accelerate the inference of many libraries and devices. This type of quantization has many properties. As QAT requires access to the entire training set, it inevitably increases the risk of data exposure. PTQ, however, needs only a small amount of calibration. It does not need the training dataset except for a minimal amount of data that can be constructed by randomly sampling instances from the calibration data. Privacy concerns still exist but are reduced compared to QAT.

Also, post-training quantization is network architecture-free, back-propagation free, and does not require domain knowledge or optimization tricks. However, it has the problem of significant accuracy degradation, especially when both activations and weights are quantized into very low-bit integers. Bai et al. in [35] analyzed different experiments to

evaluate QAT and PQT. Experiments show that QAT usually maintains better-quantized performance than PTQ. The performances of QAT are close to full-precision fine-tuning results. However, the performances of PTQ drop significantly. This accuracy loss is unacceptable in many real-world use cases.

For example, in [36], authors used object detection models for autonomous vehicles. Embedded cameras and various sensors quickly detect vehicles, pedestrians, traffic lights, traffic signs, and other objects around the cars to ensure driving safety. The object detection model should satisfy the following two conditions: first, the high detection accuracy of road objects is needed. Secondly, a real-time detection speed is essential for the detector can be used in driving. Quantization techniques involve low-precision floating-point or integer-format instructions to reduce the inference latency and the DNNs model's size. However, quantization is still defined as a lossy model compression technique, and re-training is always required to recover the precision degradation.

2.3.1.1 Pruning

Neural network pruning consists in removing irrelevant and redundant weights from a trained model. Pruning is one of the most important techniques to reduce a large NN into a smaller NN without retraining. Unnecessary weights are pruned away to yield a compact representation of the effective model[37]. DNNs are usually over-parameterized[38] with significant redundancy in the number of required neurons, resulting in unnecessary computation and memory usage at inference time. Pruning feature maps results in running networks more efficiently and speeds up inference. The early work in this domain aimed to reduce the storage requirement of the DNN model and make it storage friendly.

Ardakani et al. [39] proposed to randomly remove some of the connections in fully connected layers and proved that this method improves network accuracy while removing up to 90 % of connections. Pruning aims to reduce DNNs complexity and avoids over-fitting. A recent pruning method consists in removing filters that are proven to have a negligible impact on the final accuracy of the network. The pruning automatically removes the filter's corresponding feature map and related kernels in the next layer.

Pruning methods are roughly categorized into two main classes: structured and unstructured. Structured pruning means pruning a more significant part of the network, such as a channel, a layer, or an entire convolutional filter. This pruning removes structured DNN parts to compress and speed up DNNs. It changes the input and output shapes of layers and weight matrices, thus permitting dense matrix operations. However, aggressive structured pruning often leads to significant accuracy degradation.

Some prior works introduce pruning of the convolution layer (channel-wise, kernel-wise, and intra-kernel stridden sparsity) at different scales. The proposed method in [40] uses a specific filtering approach that helps locate pruning candidates. After pruning, fixed-point optimization (4-bit and 5-bit precision) is applied to reduce the model size further and make the model on-chip-based implementation friendly for embedded devices.

Lin et al.[41] proposed an efficient structured pruning method for jointly pruning fil-

ters and other structures in an end-to-end manner. With unstructured pruning, neurons with small saliency are removed wherever they occur. Specifically, the authors proposed an iterative approach using generative adversarial learning to learn the sparse soft mask, which forces the output of specific structures to be zero. However, this approach provides a model with high sparsity, which resulted in the increased complexity of hyperparameter optimization.

Signorini et al. [42] utilized the pruning method to remove parameters. The first step is to learn the connectivity of the trained NN, i.e., to know which parameters are more important than the others. The next step consists in pruning those connections with weights below a threshold, i.e., converting a dense network into a sparse one. Further, the important step of this method is to fine-tune the network to learn the weights of the remaining sparse connections. If the pruned network is not retrained, then the resulting accuracy is considerably lower [43].

Zhang et al. [44] proposed a framework for systematic weight pruning of DNNs using the alternating direction method of multipliers (ADMM). First, they formulated the DNN weight pruning problem as a non-convex optimization problem with combinatorial constraints specifying sparsity requirements. It was then subjected to systematic weight pruning using the ADMM framework. The original non-convex optimization problem is decomposed into two sub-problems solved iteratively. In the weight pruning problem, one of these sub-problems can be solved using stochastic gradient descent, and the other can be solved analytically.

Numerous methods have been proposed to determine the weight zeroizing criterion, such as Hoffman code [34]and iterative thresholding selection [42]. In recent years, pruning is also used to reduce the computation and speed up the inference process by pruning parameters/filters from the convolutional layer. In [45], pruning the filter reduces the number of MAC operations in the convolutional layer, and reduction in the MAC operations improves the inference time. However, finding the optimal number of parameters that can be pruned without significantly affecting the model performance is time-consuming and requires iterative retraining.

In the same direction, authors in [45] propose to remove filters that have a small contribution to the final accuracy of the network. Pruning allows the elimination of the filter's corresponding feature map and related kernels in the next layer. The relative importance of a filter in each layer is measured by calculating the sum of its absolute weights. At each iteration, the filters with the smallest values are pruned. Re-training the reduced network is required to recover the performance degradation due to the filter-removal step.

Pruning methods eliminate 10 to 30 percent of the network's weights. The network size decreased with pruning, but it is required to retrain the network to avoid change or a significant drop in accuracy. In addition, all pruning criteria require manual setup of sensitivity for layers, which demands fine-tuning of the parameters and could be heavy for some applications. Finally, network pruning can usually reduce the model size. However, it is a complex approach that impacts performance, may require retraining, and does not allow running a NN at the edge [46].

2.3.2 Knowledge Distillation

Knowledge distillation refers to transferring knowledge from a large, complex model or set of models to a single smaller model that can be practically deployed under real-world constraints. KD is based on the Teacher-Student concept. A teacher is an original model trained for a specific task [47]. It is used to teach a compressed or replicated version of itself, referred to the student. The student is encouraged to mimic the teacher output distribution, which helps the student to generalize much better and, in some instances, leads the student to perform better than the teacher.

Knowledge typically refers to the learned weights and biases. Among different model compression schemes, KD has received much attention because of its great flexibility in teacher-student network architectures. At the same time, the sources of knowledge in a large DNN are diverse. Typical knowledge distillation uses the logits as the source of teacher knowledge, while others focus on the neurons or activations of intermediate layers. Other relevant knowledge includes the relationship between different types of activations and neurons or the parameters of the teacher model themselves.

Authors in [48] have addressed the problem of making the student directly mimic the teacher's feature in the penultimate layer. Distilling features in the middle layers suffer from the different architectures between teacher and student, while transforming the features may cause the loss of some information in the teacher. Zagoruyko et al. [49] use feature matching loss to facilitate knowledge transfer. They used an attention-based distillation method to match the activation-based and gradient-based spatial attention maps.

Feature-based Knowledge has been adopted in [50] to allow the training of a student that is deeper and thinner than the teacher. FitNet made the student mimic the full feature maps of the teacher to learn from the intermediate representations of the teacher network. This approach improves the training process and the final performance of the student. The second form of Knowledge is *Response-Based Knowledge*.

Response-Based Knowledge refers to the response of the last output layer of the teacher model. The main idea is to mimic the teacher model's final prediction directly. The response-based knowledge distillation is simple yet effective for model compression and has been widely used in different tasks and applications.

Some works used response-based Knowledge to learn compact and fast object detection networks with improved accuracy [51]. They focus on transferring Knowledge within the same domain (images of the same dataset) with no additional data or labels. They are opposed to other works that might rely on data from other fields (such as high-quality and low-quality image domains or image and depth domains). Both response-based and feature-based Knowledge use the outputs of specific layers in the teacher model.

Relation-based knowledge further explores the relationships between different layers or data samples. This knowledge that captures the relationship between feature maps can also be used to train a student model. Bergmann et al.[52] introduce a framework for unsupervised anomaly detection based on student-teacher learning. In this work, several student networks are trained to regress the output of a descriptive teacher network that was pre-

trained on a large data set. Anomalies are detected when the outputs of the student networks differ from that of the teacher network, and the intrinsic uncertainty in the student networks is used as an additional scoring function that indicates anomalies.

KD can be executed directly after teacher pre-training [53] or after teacher fine-tuning. There are two ways of KD: *offline KD* consists of pre-training the teacher first and then fixing it, meaning that the knowledge can only be transferred from the teacher to the student. However, the *online KD* methods are more attractive because the training process is simplified to a single stage, and all the networks are treated as students. Unlike the offline KD between a static pre-defined teacher and a student, an ensemble of students learn collaboratively and teach each other throughout the training process [54].

Theoretically, a more robust teacher provides constructive knowledge and supervision to the student. Consequently, the intuitive approach for learning a more accurate student is to employ a bigger and more robust teacher. However, experimental results in [55][56] show that a large and robust model only sometimes makes a better teacher. As the teacher's capacity grows, the student's accuracy rises to some bound and then drops. Two crucial reasons explain this experimental results [57]. First, in some cases, the student cannot follow the teacher due to the large model due to the gap between capabilities of the teacher model and the student one. Second, the student can follow the teacher but cannot get valuable knowledge from the teacher, indicating a mismatch between the KD losses and accuracy evaluation methods.

KD allows to get smaller models, but these models are not efficient enough when the teacher model is large. Thus, KD cannot be considered as the best solution for large DNNs inference on edge. KD-based approaches can make deeper models help significantly reducing the computational cost. However, there are few drawbacks. One of those is that KD can only be applied to tasks with softmax loss function, which limits its usage. Another drawback is that KD-based approaches generally achieve less competitive performance compared with other type of approaches [46].

The following subsection contains another compression method different from the previously presented methods: *low-Rank Factorization*.

2.3.3 Low-Rank Factorization

Low-rank factorization is one of the techniques used for NN compression. The low-rank matrix approximation approximates a matrix by one whose rank is less than that of the original matrix. The goal of this approach is to obtain more compact representations of the data with limited loss of information. The weight matrices in a neural network are often low-rank, indicating redundancy in model weights [58]. Thus, the main idea is to factorize the weight matrices into smaller matrices. This technique can preserve much of the information while reducing the number of parameters.

The factorization of the dense layer matrices mainly improves the storage requirement and reduces the number of parameters. Feed forward weight matrices can be factorized into lower rank matrices to reduce the number of parameters [58]. In the same direction,

authors in [59] replace a layer in a neural network with two layers whose weights are low-rank factors of the original layer's weight tensor. Low-rank factorization reduces the number of parameters and multiply-add operations (MACs). Since factorization is restricted to those that can be realized as individual layers, the potential for compression is limited. The low-rank approximation of a matrix appears in many prior works.

Denton et al. [60] use a low-rank approximation to reduce the number of computations in the convolutional and the FC layer. They exploit the redundancy present within the convolutional filters to derive approximations that significantly reduce the required computations. Low-rank tensor decomposition can be used in CNNs to increase the network's inference speed and reduce the network's size by removing a significant amount of redundancies in the convolutional kernels [61]. Reducing the network size is crucial for deploying neural networks in mobile devices due to memory limitations and bandwidth and latency constraints. In other research, authors Zhang et al.[62] used low-rank factorization to minimize the reconstruction error of the nonlinear responses, which helps to reduce the complexity of filters and computations.

Other works used low-rank decomposition and knowledge distillation to decompose a model into small matrices, preserving much of the information while reducing the number of parameters. For example, Noach and Goldberg [63] introduced a two-stage approach to compress a pre-trained model. They decompose each weight matrix in the pre-trained model in the first stage. Then, they fine-tune or use knowledge distillation to refine the weights for the second stage. Matrices decomposition in the NN model refers to layer partitioning into small weight matrices, which is a part of the work presented in this thesis. Low-rank factorization-based approaches are straightforward for model compression and acceleration. However, the implementation is challenging since it involves decomposition operation, which is computationally expensive.

Another issue is that current methods perform low-rank factorization layer by layer and thus cannot perform global parameters compression, which is essential as different layers hold different information. Finally, factorization requires extensive model retraining to achieve convergence compared to the original model, while this thesis mainly focuses on DNNs inference and avoids the retraining step.

This section shows that prior works used model compression techniques to compress models and reduce the number of parameters which is important in enabling large DNNs on edge devices. Meanwhile, previous works [64] assert that different compression techniques produce consistently less robust models than the large uncompressed model. Moreover, current compression techniques still largely depend on human heuristics to achieve good performance. For example, pruning relies on the saliency score.

KD often requires a designed loss function, weight sharing, and low-rank factorization involving expertise to appoint modules for sharing or factorization [65]. Most model compression techniques modify DNNs structure and demand retraining and fine-tuning. However, the main goal of this thesis is to focus on the inference of large models with high accuracy and avoid retraining.

According to the criteria grid mentioned in subsection 2.2.2, the existing model compression approaches are classified and compared in Table 2.1. All model compression ap-

proaches cited in Table 2.1 reduce the model size to fit on the edge device. The modifications made to the original model's structure often **cause an accuracy loss**. The compressed model can be executed on **a single device** in one block. Thus, these approaches **do not adopt any partitioning strategy or scheduling policy**.

Existing works	Model structure changes	Model re-training requirement
[28],[30], [34],[35], [36]	Quantization	✗
[31],[32]	Quantization	✔
[39], [41], [42],[45]	Pruning	✗
[49],[50] , [55],[56]	Knowledge Distillation	✔
[60],[61], [65],[62]	Low-Rank Factorization	✗

Table 2.1: Summary of existing model compression approaches for adapting DNNs to run at the edge.

✔: means the proposed approach does not require a re-training phase.
✗: means the proposed approach requires a re-training phase.

2.4 Partition and Distribution of DNN Models at the Edge

Model compression-based approaches presented in section 2.3 are used to enable DNNs structure to be adaptable to run on resource-constrained devices. However, for large and complex DNNs, these techniques are not enough to deploy models on edge while preserving high accuracy and avoiding retraining. Applications that include DNNs require high accuracy, thus, the topology of DNNs evolves and becomes more complex with huge parameters.

With an increase in IoT technology and the rise of the edge computing paradigm, several edge servers are located in the continuum between end devices and the cloud. Edge servers bring extremely powerful connectivity to edge computing with low latency and high processing speed. So, several existing works aim to exploit edge servers(servers, routers, containers, hubs) to perform DNNs inference.

However, edge servers suffer from limited computation resources and low memory. DNN partition consists of splitting a DNN structure into several small partitions and run different partitions in different edge nodes. DNNs partitioning aims to reduce the gap between large computing workloads of DNNs and limited computing resources of edge devices.

DNNs topology[66] [67]is categorized into Directed Acyclic Graph (DAG) topology DNNs and chain topology DNNs. Thus, two strategies are commonly adopted to partition DNN according to its topology. The first strategy considers the DNN as DAG topology and exploits graph partitioning techniques to split it. The second strategy is to partition the DNN model based on the particularity of its architecture. In this section, existing works that use generic partition strategies are presented then we focus on previous works that discuss typical-based DNNs partitioning approaches.

2.4.1 Generic Partitioning Strategies

To partition DNN models, a lot of research consider DNN structure as a graph and use graph partitioning techniques to split the NN [68][69]. Some complex DNNs are characterized by DAG topology, for example GoogLeNet and ResNet. Other chain DNNs can be converted to DAG topology DNNs[67]. In a DAG representation of a DNN, each node(vertex) represents a layer and edges represent the data communication between two nodes. DeepSlicing [70] models CNNs as DAGs then partition it into multiple blocks according to specific parameter.

IONN [71] proposed a partitioning-based DNN technique for edge computing. The DAG contains multiple paths. A path is a sub-graph of the DAG with a line structure that starts from the input layer and ends at the output layer. For general-structure DAGs, the partition could spread across different paths. IONN considers a chain topology DNN as a DAG and splits the DNN into partitions by iteratively finding the fastest execution path on the graph.

Authors in [72] introduce Dynamic Adaptive DNN Surgery (DADS) to enhance the layer-wise partition to complex DNNs represented by DAGs. This work aims to find the optimal splitting point. A splitting point is a location in a NN architecture that is decided to be a border across two partitions. DADS employs edge computing and deploys DNN layers to an edge node and a cloud server. However, DADS cannot generalize the partitioning approach to separate a DNN into more than two partitions.

Some existing solutions focus more on scheduling by targeting a well-defined distribution policy (Edge-Cloud, Mobile-Edge-Cloud, etc.). The scheduling then consists of a sequence of tasks ending in the cloud.

Zhang et al. in [73] introduces a heuristic algorithm to split a DNN according to per-layer processing time and inter-layer transmission delay into three parts. The DNN is modeled as a DAG then splitted into three sub-graphs by assigning each vertex to one of the three computing tiers and minimizing the total latency. The three parts are executed over device, edge and cloud.

Authors in [74] introduced the edge-cloud computation offloading problem into a graph min-cost partitioning problem, in which computation tasks are optimally distributed between mobile devices and cloud. The proposed min-cost offloading partitioning algorithm took both the execution time and energy consumption into account when deciding an optimal task partitioning positioning. In [75] Hu et al. introduced EdgeFlow, a new distributed inference mechanism designed for general DAG structured deep learning models. EdgeFlow partitions and distributes the model among different devices. In [76], authors consider

a DNN model a DAG to formulate the fine-grained relationship between different neurons inside logical layers and partition the model into multiple partitions.

A lot of existing works on distributed inference field consider the model with a chain structure, which strongly hinders the applicability since most modern DNNs are constructed as complicated DAGs. The adaptation to the DAG structure is non-trivial, and the challenges can be summarized in the following two aspects. In distributed inference, it is important to keep the layer dependencies that indicate the proper execution order of the layers and allow to obtain the correct results. With a chain structure, the dependency of the layers is very straightforward as one layer only has one preceding and one succeeding layer, respectively.

However, in a DAG structure, one layer requires the results from multiple preceding layers as input or may be needed by multiple subsequent layers. Compared with the chain structure, the layer dependencies inside a DAG are much more complicated, adding complexity to ensure the correct execution, especially after the layers can be partitioned and distributed across different devices. There are several existing works that proposed partitioning strategies adapted specifically to DNNs with chain topology. Some examples will be discussed in the next paragraph.

2.4.2 Typical-based DNNs Partitioning Strategies

A lot of previous research adopt partitioning strategies based on the characteristics of NN structure to allow large DNNs inference across device-edge-cloud network infrastructure. *Typical-based partitioning strategies* can be classified into *vertical partitioning* and *horizontal partitioning*. Vertical partitioning consist in splitting a DNN structure per set of contiguous and entire layers while horizontal partitioning consists in splitting DNN layers into many sets of neurons/units.

DNN structure can be deployed at the edge in several ways: (1) Cloud-Device, (2) Edge-Device, (3) Cloud-Edge-Device or (4) Device-Device. In literature, model partitioning can be applied either to offload DNN tasks across device-edge-cloud infrastructure or to distribute DNN inference across edge infrastructure. Many existing approaches are focused on task placement using simple scheduling consisting of a sequenced execution of two to three partitions.

2.4.2.1 Computation Offloading across Device-Edge-Cloud

Computation offloading refers to transferring intensive computational tasks to a remote server such as an edge server or a cloud. Computation offloading is an effective way to enhance user service quality [77] because DNNs inference is a very computationally-intensive task. Industry and academia propose computation offloading as a promising solution for effectively integrating resources in computing processes designed based on the Device-edge-cloud paradigm.

The offloading of computation-intensive tasks can significantly reduce response time

and improve the overall performance of DNNs inference. Cloud computing still has some limitations, such as high transmission costs and privacy concerns. Offloading migrates computation to the edge server, which is close to users. It could be the best approach to solve the problems related to the calculation at optimal times. It is efficient to offload the computation from end devices to the edge server: the end devices will send their data to a nearby edge server and receive the corresponding results after server processing [78].

The computation offloading policy requires to identify three elements [79]: first, when to offload, it is important to fix the time for offloading under different constraints. Second, where to offload, it is necessary to find the best placement to compute the offloaded workload according to available resources. Third, the offloading policy must have a specific objective. Offloading decision is usually related to an optimization objective to overcome resource limitations of edge devices, energy consumption, and high processing latency. Available computing resources are distributed in cloud servers, edge servers, and edge devices. According to these different locations in the network, existing works mainly focus on three offloading strategies:

Device-to-Cloud (D2C) Offloading:

D2C offloading involves transferring computations from the resource-limited IoT device(s) to resource-rich cloud centers to improve the execution performance of AI applications. Existing works proposed techniques to identify an optimal partitioning point based on the characteristics of the layers of a DNN and operational conditions, such as resource utilization or infrastructure network conditions.

For example, Neurosurgeon [80] proposed to partition model between cloud server and mobile device according to the network situation. Neurosurgeon is one of the first works to investigate layer-wise partitioning, also named vertical partitioning. The split point is decided intelligently depending on the infrastructure network conditions and devices capacities.

Authors in [81] proposed a CNN splitting algorithm that efficiently splits CNN vertically in exactly two parts, between edge and cloud and reduces bandwidth consumption. Various parameters are considered, such as CPU/RAM load at the edge, input image dimensions, and bandwidth constraints, to choose the best splitting layer.

Device-to-Edge server (D2E) Offloading:

D2E refers to offloading computations from resource-constrained devices to edge servers close to the end devices at the network's edge. D2E offloading addresses the high latency issue in delay-sensitive services and applications that are not properly handled within the Cloud computing paradigm. In [82], authors propose an edge computer offloading technique that assigns computational tasks generated by devices to potential edge computers with enough computational resources. This approach is based on edge computers clustering depending on their hardware specifications. Afterward, the tasks generated by devices will be pushed to a hybrid ANN model that predicts the profiles, i.e., features, of the edge computers with enough computational resources to execute them.

Li et al. [83] present the framework Edgent that leverages edge computing for DNN collaborative inference through device-edge synergy. This framework permits the distri-

bution of DNN computations between mobile devices and the edge server according to the available bandwidth. In [84] authors proposed an energy-efficient autonomic offloading scheme that can automatically offload computational tasks to edge servers. This work aims to minimize the total energy consumption of applications running on a mobile device.

In [85], authors present Hierarchical Machine Learning Tasks Distribution (HMTD) a framework for a target tracking system that aims to minimize the weighted-sum cost with the inference error rate constraint. In the proposed framework, a trained CNN is divided vertically into two parts: lower-level layers and higher-level layers. The lower-level layers of the deep learning model are implemented at the unmanned aerial vehicle (UAV), while the higher-level layers are deployed at the multi-access edge computing (MEC).

Device-to-Device (D2D) Offloading:

Edge devices play the role of data producers and data consumers. There is great potential via collaboration and cooperation among the devices, besides offloading their computational tasks to more powerful ends. Thus, data generated at the edge will be processed locally at the network's edge instead of transmitting to the cloud or edge servers.

Authors in [86] proposed a distributed D2D offloading system that can guarantee a high probability of on-time task completion and low energy consumption. If one edge device is not powerful enough to provide a real-time response for model inference, a cluster of edge devices could cooperate and help each other to give enough computation resources. For example, if a camera needs to perform an image recognition task, it could partition a CNN model by layers and transmit the partitioned tasks to other devices nearby.

Partitioning a CNN model by layers means splitting the CNN structure vertically to obtain partitions that include consecutive layers. The decision of which task to offload is the first and most significant challenge to address, as it comprises the core of the task offloading problem. This decision is mainly based on a vertical partitioning strategy to decide whether the task should be executed locally or offloaded to a remote infrastructure.

A failed partitioning strategy may result in performance bottlenecks regarding the execution of the application. In [87], Chen et al. proposed a novel D2D framework where a massive cluster of devices shares computation and communication resources to achieve energy-efficient collaborative task executions. In the same direction, authors in [88] leverage the software agents running on the IoT devices to establish an integrated multi-agent system (MAS). By sharing data and information among mobile agents, edge devices can collaborate and improve the system's energy efficiency in executing distributed applications.

The offloading decision is the key element for the offloading frameworks. Some frameworks take the offloading decision at runtime based on program profiling and program analysis, while others take the decision during design or compile time using estimations. Most offloading strategies are based on a partitioning step that aims to pick which parts of an application's execution to retain on the edge device and which to migrate to the cloud or edge server. Not all existing works require partitioning. Various techniques need cloning the running application to the edge server or cloud [89]. The choice of the best way of partitioning depends on several factors, such as available edge infrastructure, offloading objective, and complexity of AI applications.

Recent research on edge computing found that applying DNN to end devices will bring great convenience to people and the industry. Indeed, applications that include DNNs become more complicated due to massive data collected by IoT devices at the edge and high computation overhead. Therefore, it is recommended to avoid data transmission to the cloud and run the DNN model locally to accelerate edge DNN inference so that data can be transmitted at high speed in a relatively safe infrastructure. The following subsection presents a literature review of research works on DNN partition and deployment at the edge.

2.4.2.2 Distributed DNN across Edge Devices

D2D offloading refers mainly to distributing computations over edge devices by sharing tasks with nearby devices or cloning tasks over multiple devices. D2D offloading facilitates computation resource pooling and sharing among edge devices. However, the D2D approaches do not focus on DNN partitioning. This subsection concentrates specifically on distributing DNN structure across edge devices.

Partitioning the DNN structure is an interesting research field among various works in this area of research, and there are a few works done in this area. Previous works aim to partition and distribute DNN inference between edge devices. Model partitions are deployed separately on devices at the edge. Zhao et al. [90] proposed DeepThings, a locally distributed and adaptive CNN inference framework in resource-constrained IoT devices. DeepThings proposes a Fused Tile Partitioning (FTP) which consists in partitioning all convolutional layers horizontally into independent tasks, allowing the parallel execution of the distributed inference. Deepthings proposed a scheduling policy to reuse overlapped data between adjacent CNN partitions.

In the same direction, DeepSlicing [70] considers the varieties of the model structure. Still, it requires finding specific points to split the model into two sub-models that can be executed sequentially. DeepSlicing splits feature maps along the longer dimension of height and width. This data partitioning approach aims to avoid redundant computations resulting in reducing the inference latency. Moreover, MoDNN [91] partitions layers horizontally and the layers' input and output data using the Biased One-Dimensional Partition (BODP) method. MoDNN treats each computing part of every single layer as an individual task, leading to high synchronization costs among devices. The mutual waiting would also greatly increase the inference latency. This approach focuses mainly on sparse fully-connected layers (i.e., fully-connected structures where some weights are zero). In this work, weight-intensive convolutional layers are not addressed.

On contrary, authors in [92], partitions fully-connected layers, as well as feature and weight-intensive convolutional layers, to serve a wide range of CNN inference tasks. By integrating the horizontal partitioning strategy, which is named *Weight partitioning*, with a communication-aware layer fusion approach, holistic optimization across layers was achieved, allowing memory and computation demands to be optimized simultaneously. The proposed method allows for the complete distributed execution of a CNN application across a cluster of resource-constrained edge devices. Table 2.2 contains a comparison of different existing frameworks.

Frameworks	Partitioning type	Data Partitioning method	Scheduling granularity	General CNN
DeepSlicing [70]	Vertical partitioning, data partitioning	One-dimension	Arbitrary layers	Yes
MoDNN [91]	Horizontal partitioning, data partitioning	One-dimension	Layer	No
DeepThings [90]	Horizontal partitioning, data partitioning	2D-dimension	CNN	No

Table 2.2: Comparison of different Frameworks [70]

Yang et al. [93] proposed CoopAI, a collaborative edge computing system that makes use of multi-layer partitioning. This system allows multiple layers to be grouped into a block to process multiple layers in a round. Edge devices that work together on multiple layers yet require intermediate results from each other.

In [94], Zhou et al. introduced a containerized partition-based CNN inference at the edge. This framework dynamically partitions a DNN model using horizontal partitioning techniques. The output feature maps can be partitioned along the channel dimension such that each device computes a subset of the output feature maps. The proposed framework containerizes and deploys each partition on a small cluster of IoT devices using Kubernetes for better resource management and scheduling.

Authors in [95], introduce DistrEdge, a CNN inference distribution method that models the split process as a Markov Decision Process and utilizes Deep Reinforcement Learning to make optimal split decisions. DistrEdge slits a CNN architecture vertically into parts that contain one or more layers. DistrEdge can adequately adjust the distribution strategy according to the device computing characteristics and the network conditions.

In [96], EdgeSP adopts a multiple-fused-layer-block parallelization strategy to reduce the communication overhead between devices during parallel inference. This approach effectively reduces the average task inference delay and improves resource utilization by adding early exit branches.

Horizontal partitioning allows for the reduction of memory footprint and computation complexity. However, this strategy incurs high communication overhead due to frequent data movement among partitions. In addition, a fusion step is required to generate a complete feature map. Another existing partitioning strategy is to split a DNN model vertically per layer. This strategy supports the execution of partitions in a pipeline fashion, which may help to exploit both data-level and task-level parallelism and to increase the system performance. Vertical splitting aims mainly to reduce the memory needed for data storage[21].

Compared to the vertical partitioning strategy proposed in [21], authors in [97] used approximate computing techniques to fit CNNs into tiny embedded systems. These embedded systems run on ARM big.LITTLE Multi-Core processors [98] by partitioning CNN

layers across heterogeneous cores to improve throughput. They implement a sequential vertical partitioning strategy which consists of partitioning a CNN model into partitions that include consecutive CNN layers. The memory required to deploy each partition can be significantly reduced. Both the memory needed for storing the data exchanged between the layers of a partition and the memory required for storing the weights of the layers of a partition can be significantly reduced when the number of partitions is large. For large CNN models, more than the vertical partitioning granularity is needed to get V-partitions that fit with resource-constrained devices.

Authors in [99] presented DEFER as a framework to sequentially partition the DNN model into smaller partitions across multiple edge devices, such that each device sends its computed inference result to a subsequent device. DEFER adopts layer-wise splitting, which refers to vertical partitioning to increase throughput and decrease per-device compute load. Each compute node takes charge of the computation of a specific partition and then sends a compressed result to the following compute node. DEFER aims to compare inference throughput for different serialization, and compression configurations exchanged between compute nodes. This work measures energy consumption and does not consider the communication overhead.

In [100], Distributed INference Acceleration (DINA) proposed a fine-grained adaptive horizontal partitioning scheme to divide a source DNN into partitions that can be smaller than a single layer. These partitions can be processed locally by end devices or offloaded to one or multiple powerful nodes, such as in fog networks. Partitioning considers the specific characteristics of layer types in commonly-used DNNs through an efficient matrix representation that reduces the communication overhead in the network.

Table 2.3 presents a classification of some DNNs partitioning frameworks according to their partitioning strategy and performance parameters. All the examples given in this are then included in the summary table in the last section.

Framework	**DNN Partitioning**	**Improvement objective**
DDNN [101]	vertical partitioning	latency and energy consumption
DeepThings [90]	data and horizontal partitioning	latency and memory footprint
jointDNN [102]	vertical partitioning with multiple splitting points	latency and energy consumption
DeepSlicing [70]	vertical partitioning	latency
Neurosurgeon [80]	vertical partitioning	latency

Table 2.3: Summary of DNN Partitioning frameworks

Vertical partitioning is not enough to deploy large DNNs inference on resource-constrained devices. Horizontal partitioning incurs communication and synchronization overhead. The partitioning strategies adopted by previous works can change layers structure to reduce

memory and computation demand resulting in model accuracy loss and increased communication overhead. Our contribution in this thesis is to use the existing partitioning strategies in a smarter way. The proposed solution allows to split a large DNN model into small partitions without modifying the main model structure while avoiding accuracy loss and minimizing communication overhead.

2.4.3 DNN Scheduling for Distributed Inference

Defining scheduling policy is essential to perform partitioned model inference on distributed infrastructure. Scheduling consist in ordering the execution tasks and assigning each partition's placement. Previous works that performed distributed inference introduced a scheduling policy to overcome several issues. From a spatial perspective, scheduling ensures that all model partitions are distributed across the target devices.

Also, scheduling allows us to consider the order of partitions execution and ensure temporal arrangement throughout the execution process. Hu et al. in [103] use scheduling to efficiently schedule the tasks, by using a queuing time-aware scheduler. The scheduler allows management of the whole cluster's metadata and sharing the device status with all the devices in the cluster. Partitioning and scheduling are correlated. In [104], authors proposed a scheduling algorithm to topologically order all tasks by considering precedence, application, and resource constraints. Through distributed inference process, scheduling can reduce energy consumption and inference response time[105].

Furthermore, Deepthings [90] employs a scheduling process to improve data reuse and identify the overlapped region in the different model partitions. The scheduling defines work items in a separate stealing order that minimizes dependencies and thus maximizes parallelism. In our contribution, scheduling is related to the manager that uses the communication hub to allow the distribution of model partitions between devices and the data transmission during the inference process. Scheduling allows efficient, timely distribution of partitions to target devices and minimizes manual intervention by automating the inference process.

2.5 Synthesis and Conclusion

Based on the criteria grid defined in Section 2.2.2 and inspired by a synthesis table in [106], Table 2.4 includes a summary of prior works mentioned in the previous sections and our proposed solution (namely HyPS). This table highlights the strengths and limitations of both existing approaches and our proposed solution. To ease the understanding of Table 2.4, we add symbols that indicate whether it is a weak or strong point of the proposed approach.

- Yes ✗ or No ✗: refers to a limitation on the proposed approach.
- Yes✔or No✔: refers to a positive aspect of the proposed approach.

The Edge computing paradigm provides low latency, mobility, and location awareness support to delay-sensitive applications. This chapter showed that combining Edge computing and AI technologies is an efficient way to solve the problem of deploying large DNNs on resource-constrained infrastructure. Several model compression approaches are used in the literature to adapt the DNN models structure to fit edge devices. These approaches reduced the model size and ran compressed DNNs at the edge. However, model compression approaches change the original DNNs structure, increasing the accuracy loss and, in some cases re-training the model to recover this loss.

Also, existing approaches propose to either offload DNN inference workloads to the cloud or to handle the workload within the resource-constrained devices using various innovative techniques. Significant research has been carried out to partition the DNN structure into small partitions using multiple partitioning strategies. Distributed inference of partitioned DNN requires the collaboration of multiple edge devices. There are several limitations to these partitioning strategies approaches:

- Re-training the DNN model may be required to ensure high performance,
- For data loads, significant network bandwidth may be required,
- Privacy may be impacted because user data is no longer on personal devices,
- Some partitioning strategies and scheduling policies give rise to communication overhead.

Existing works	Model structure changes	Accuracy loss	Model retraining requirement	Partitioning type	Partitions placement	Running on single edge device
[28],[30], [34],[35], [36]	Quantization	Yes(✗)	Yes(✗)	N/A	edge	Yes(✔)
[31],[32]	Quantization	Yes(✗)	No(✔)	N/A	edge	Yes(✔)
[39], [41], [42],[45]	Pruning	Yes(✗)	Yes(✗)	N/A	edge	Yes(✔)
[49],[50] , [55],[56]	Knowledge Distillation	Yes(✗)	No(✔)	N/A	edge	Yes(✔)
[60],[61], [65],[62]	Low-Rank Factorization	Yes(✗)	Yes(✗)	N/A	edge	Yes(✔)
Neurosurgeon [80] Deepsplit [81]	No(✔)	No(✔)	No(✔)	Vertical partitioning	device-edge-cloud	No(✗)
DeepThings [90]	No(✔)	Yes(✗)	Yes(✗)	Data partitioning , horizontal partitioning	edge	No(✗)
Deeperthings[92]	No(✔)	N/A	No(✔)	Data partitioning, , horizontal partitioning	edge	No(✗)
DINA [100]	No(✔)	No(✔)	No(✔)	Horizontal partitioning	Device-edge	No(✗)
CoopAI [93]	No(✔)	N/A	No(✔)	Vertical partitioning	edge	No(✗)
MoDNN[91]	No(✔)	No(✔)	No(✔)	Data partitioning, Horizontal partitioning	device-edge	No(✗)
DeepSlicing [70]	No(✔)	No(✔)	No(✔)	Data partitioning, Vertical partitioning	edge	No(✗)
Our solution (namely HyPS)	*No(✔)*	*No(✔)*	*No(✔)*	*Hybrid partitioning*	*edge*	*Yes(✔)*

Table 2.4: Summary of existing approaches for enabling DNNs inference at the edge

Chapter 3

Hybrid Partitioning for CNNs Inference at the Edge

3.1 Introduction

Existing partitioning strategies proposed in chapter 2 enable large CNNs partitioning. Most of them distribute the execution of these partitions across multiple devices or offload them to the cloud. Existing approaches may increase latency and communication overhead. These approaches do not enable large CNNs inference on a single device without applying compression techniques, reducing accuracy and/or requiring re-training. However, a Hybrid partitioning strategy that mixes the two ways of partitioning (vertical and horizontal) allows large CNNs inference on a single device with high accuracy and avoids cloud computing. This technique aims to split a CNN structure efficiently into several partitions that can be deployed separately on resource-constrained devices.

The main objective of this contribution is to capitalize on the CNN models previous training phase, use it actively on inference and avoid the re-training phase as much as possible. This chapter is divided into three parts. First, the proposed partitioning strategy will be presented in details. Second, the architecture of the solution is presented as well as the advantages of this contribution. Then, we highlight some of the most promising real-world use cases in which our proposed strategy can be beneficial. Conclusions and challenges are discussed at the end of this chapter.

3.2 Hybrid Partitioning Strategy

3.2.1 Problem Formulation

Inferring pre-trained large CNNs consumes significant time, memory, and computational resources that can be higher than most of the edge devices capabilities. For example, a large CNN like VGG16 cannot be deployed on Raspberry Pi 3 B with 1 GO of RAM. Apart from hardware capabilities, edge devices often suffer from failures that result in unpredictable service loss.

CNN execution on edge devices needs to be robust enough to cope with a satisfactory quality of service from the user's point-of-view. Because edge devices are often located in unprotected areas (e.g., customers' homes), another drawback of edge intelligence is related to secrets protection. This topic is tremendous owing to the risks of disclosures concerning data and the DNN models. Several approaches partition the CNN architecture into small partitions to reduce memory footprint and fill the gap between the resource demands of CNNs models and edge devices' capabilities for inference. However, they do not cover the subject as proposed in this thesis, HyPS allows to perform large CNN inference at the edge:

- Without accuracy loss, HyPS maintains precisely the same accuracy as the original DNN model running on a powerful machine.
- HyPS allows large models to fit on a single devices with limited computation and memory capacity.

- Distributed inference of partitioned DNN allows low inference latency and low communication overhead.
- HyPS does not require any re-training step,
- HyPS does not modify the original CNN structure
- Data generated at the user machine is processed locally without transmission to the cloud, which increases data protection.

Therefore, the proposed solution in this thesis brings an essential contribution to the inference deployment of complex AI models at the edge.

3.2.2 Governing Example VGG16

For the sake of clarity, a governing example has been retained to illustrate principles and how they can be applied to a realistic example. In this section, HyPS is applied on an example of CNN which is VGG16 [107] [108]. VGG16 is a well-known CNN example used as Visual Geometry Group (VGG) is a popular and clear-in-structure CNN model that includes all mainstream layer types. The VGG architecture is the basis of object recognition models.

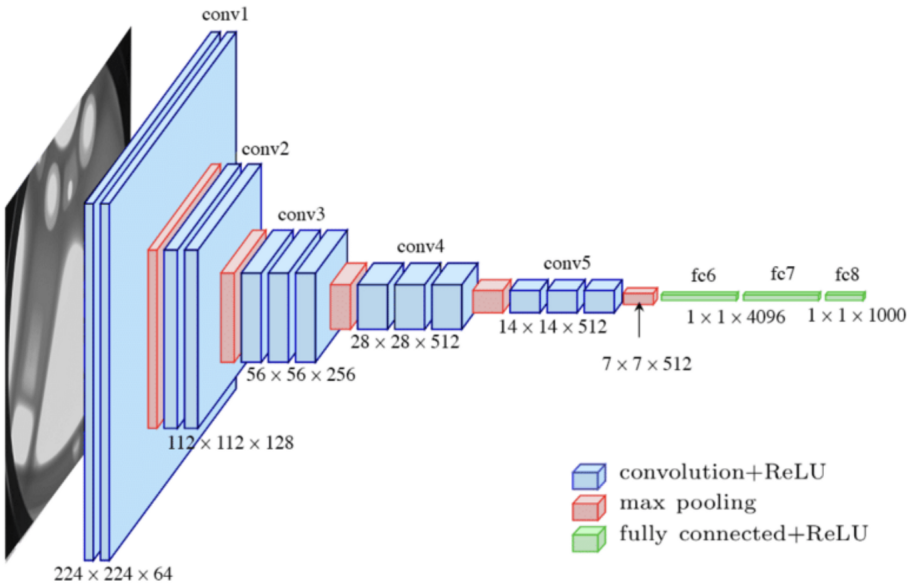

Figure 3.1: VGG16 architecture[109].

Figure 3.1 show the VGG16 structure overview. Structured as a deep neural network, the VGG surpasses baselines on many tasks and datasets beyond ImageNet[110]. VGG16 is considered as an excellent vision model architecture in classification tasks. This model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 millions images belonging to 1000 classes. The number of layers in a NN defines its depth. So the number 16 in the name VGG refers to the depth of the model. This means that VGG16 is a pretty extensive network and has a total of around 138 million parameters.

Most unique thing about VGG16 is that instead of having a large number of hyper-parameters, the VGG16 structure includes convolution layers of 3x3 filter with a stride 1 and always uses same padding and max pooling layer of 2x2 filter of stride 2. It follows this arrangement of convolution and max pool layers consistently throughout the whole architecture [111]. Table 3.1 in [112] presents the complexity and accuracy of known CNNs. MACs refers to multiply–accumulate (MAC) operations. Even according to modern standards, VGG16 is considered as a huge network.

Model	Layers	Parameters (in millions)	MACs	Error-5(%)
AlexNet	8	60	650	19.7
ZefNet	8	60	650	11.2
VGG16	16	138	7800	10.4
SqueezeNet	18	1.2	860	19.7
GoogleNet	22	5	750	6.7
ResNet-101	101	40	3800	6.8
ResNet-152	152	55	5650	6.7
DenseNet-201	201	16.5	1500	6.3
Inception-v3	48	23.6	5700	5.6

Table 3.1: Complexity and accuracy of known CNNs.

3.2.3 Vertical Partitioning Strategy

Vertical partitioning is a valuable strategy that splits a large pre-trained CNN structure so that each partition includes a set of consecutive layers. This strategy does not divide the calculated weights for each layer. Each V-partition is defined by a specific output layer and generate its own feature map.

V-partitions can be deployed separately on devices with limited memory and low computation capacity. However, to get a final inference response, it is necessary to transmit partitions and ensure feature map transmission between them. In Figure 3.2 V-partitions are represented by rectangles colored in blue and each one includes one or a group of consecutive layers.

Dimensions of feature maps produced by the output layer can vary considerably, resulting in a possible huge volume of data to transfer. Therefore, the choice of the output layer is essential. The output layer is the decisive point in the dimensionality of the generated feature map which will be transmitted to the next partition. The feature map shape through the CNN layers is irregular, and it depends on the filter size applied in the layer, the input dimension of the feature map output of the prior layers, and the type of the layer.

As mentioned in Section 2.2.1, all layers in a CNN structure are arranged following a specific pattern. The input layer of CNN is a convolutional layer, and the output layer is a fully-connected layer. The output feature map of a convolutional layer is a complex matrix with high dimensions. Therefore, after the convolution block, there is a pooling layer that

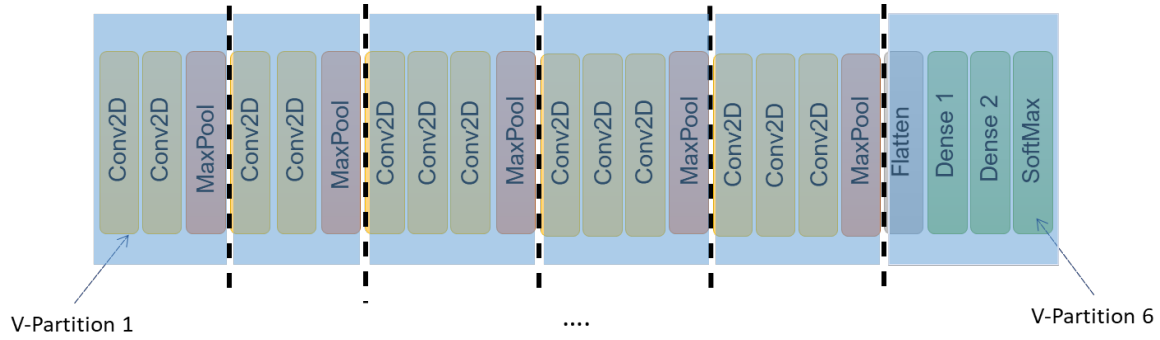

Figure 3.2: Example of V-partitioning on VGG16. Split points are depicted thanks to dashed lines.

is responsible to extract major characteristics of the data. This sort of layer performs a dimensionality reduction on the input by reducing the number of parameters. The pooling layers represent interesting splitting points which reduce the size of the transferred feature maps. HyPS uses these strategic split points to position the boundaries of the V-partitions. In Section 4.3.1 this specific points will be quantified.

Some complex layers can require a powerful computing capacity to generate feature map. For limited resources devices, it is impossible to perform calculations of complex layers in one operation. So, vertical partitioning alone is not sufficient to perform inference.

For example, experiments showed it was impossible to run VGG16 structure without horizontal partitioning on a resource-constrained device, because of a too complex layer. This specific layer has a number of parameters that exceeds one hundred two million (102 * 10^6) parameters. The total number of parameters is the sum of all weights and biases. Therefore, edge devices with limited capacity do not support the computation of massive operations at the same time. This issue implies the use of a finer level of splitting because more than vertical partitioning is needed to deliver CNN inference efficiently on edge devices. There is a need to finely split layer weights to obtain smaller partitions than those obtained with vertical partitioning. So, it requires horizontal partitioning.

3.2.4 Horizontal Partitioning Strategy

As mentioned in the previous section, the vertical partitioning splits the DNN model at the layer granularity while horizontal partitioning splits a DNN layer at neurons granularity. Horizontal partitioning is the thinnest way of partitioning.

This strategy partitions a given layer into small groups of neurons, whereas the input data layer is not partitioned. In this case, partitioning one layer into H-partitions reduces the number of parameters, storage needs, and the memory required to compute layer features.

Therefore, the previously mentioned too complex CNN layer that includes intensive computations is divided into several H-partitions. Each H-partition is only responsible for

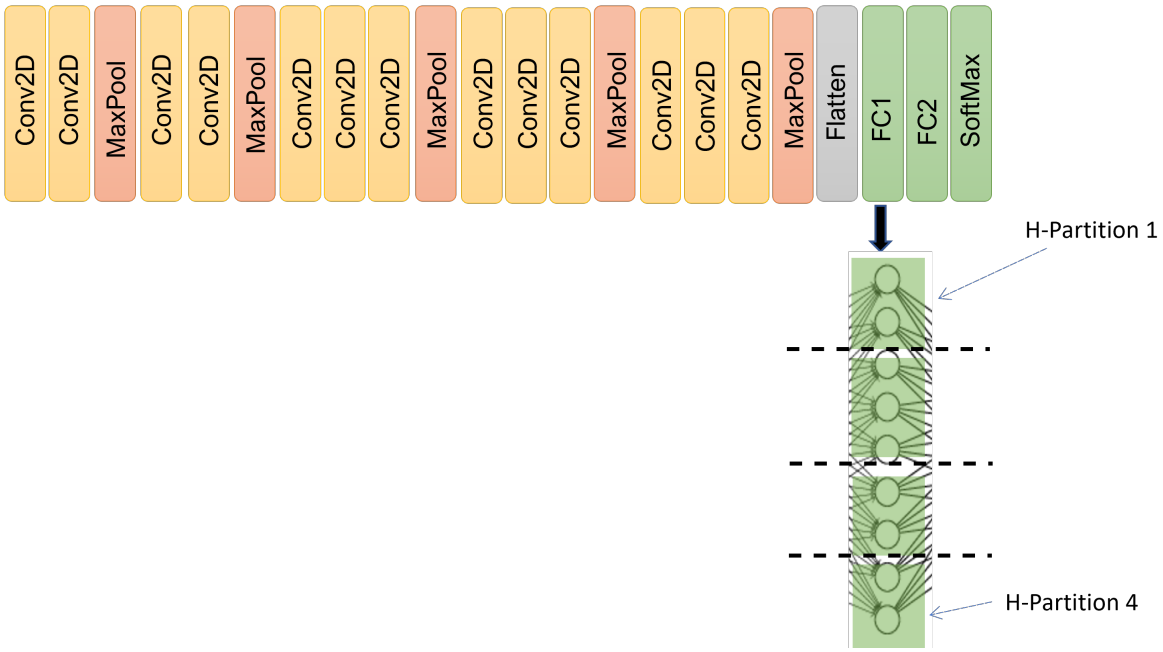

Figure 3.3: Horizontal partitioning on one layer.

computing a part of the output of the current layer. At the same time, a specific algorithm will collect and merge all the output's H-partitions to get the full feature map before executing the next layer. In Figure 3.3 H-partition are represented by green rectangle (four partitions are chosen to simplify the presentation in the figure). A partition can contain one or more neurons of the same layer and computes a partial feature map.

Partial features maps collection and merging is a process that introduces a synchronisation cost. This cost does not exist for vertical partitioning and so, to preserve performance, horizontal partitioning must be used as few as possible. The next section describes how our solution manages to take the highest benefits from both sorts of partitioning described here.

3.2.5 Proposed Strategy

Customers and industrial users usually search for an efficient and straightforward framework to execute large AI models at the edge. This type of frameworks should be based on a good partitioning strategy that enables large CNN model inference according to available infrastructure capacities. Our contribution is a decision support system that provides a guided partitioning strategy that can be applied on complex models.

This thesis proposes a *Hybrid Partitioning System (HyPS)*, a solution that comprises a hybrid partitioning strategy, an orchestration architecture, and an implementation. The hybrid partitioning strategy mixes vertical and horizontal partitioning and identifies the best positions in the model architecture to split a NN structure. Applying the hybrid strategy allows the generation of small partitions that fit the resource constraints of edge devices

noticeably by decreasing instantaneous memory needs.

HyPS takes input information about the global model structure to be partitioned and the characteristics of the target edge infrastructure. The proposed approach helps to identify the strategic split points to get partitions and precise the type of partitioning to be applied. Using either vertical or horizontal strategy could not solve entirely the problem of edge infrastructure incapacity to run a given model. For a large CNN model like VGG16 [107], vertical partitioning generates V-partitions that could still remain computationally intensive and complex to be executed on edge devices, that is why HyPS makes use of vertical partitioning with horizontal partitioning.

Horizontal partitioning splits the CNN layers into thinner H-partitions. However, H-partitions must also be fused to obtain the final result of the partitioned layer: this fusion increases both the communication and the computing times. In consequence, we argue that the number of H-partitions must be limited to avoid ineffective operations. The intended solution aims to minimize overhead costs related to H-partitions synchronization and prevent any degradation in model accuracy.

The hybrid partitioning strategy provides a solution to identify *mandatory partitioning points* and *optional partitioning points* on the CNN structure. HyPS prioritizes vertical partitioning and applies horizontal partitioning only if it is mandatory to perform the CNN partition execution on the target edge infrastructure.

Algorithm 1 Get mandatory split points

Require:

1: $Model$: CNN model

2: $Threshold$: maximum number of parameters supported by the device

Ensure: LM : list of mandatory split points

3: **for** each $layer_i$ in $model$ **do**

4: **if** number of parameters >= Threshold **then** ▷ $layer_i$ is a complex layer

5: $LM \leftarrow$ [index of $layer_{i-1}$, index of $layer_{i+1}$]

6: **else if** the last layer **then**

7: **return** LM

Algorithm 1 shows the steps to identify the mandatory partition points for an input CNN model. The proposed algorithm starts by going through all the model layers one by one and check if it is possible to run solely each corresponding layer on the target edge device. The program runs this process until reaching either the last layer or a complex one(line 4 in Algorithm 1). In the case of a complex layer, HyPS fixes two mandatory split points. For one particular complex $layer_i$: the first mandatory split point is located before the complex layer, the $layer_{i-1}$ is the output layer of the first V-partition, the second mandatory split point is located after the complex $layer_i$. So, $layer_{i+1}$ is the input layer of the second V-partition and a third constituted of the complex layer itself and alone. The output of this algorithm is a list that contains the indexes of the mandatory split points(line 5 in Algorithm 1). The mandatory split points allow to fix the input and output layers of each V-partition. The same process is applied for each fixed complex layer. Then, the V-partition containing the complex layer is itself partitioned into H-partitions as small as

required to fit targeted execution infrastructure.

The first step consists of identifying the possible mandatory split points. HyPS determines optional split points in the second step. Optional split points are specific locations in the CNN architecture where partitioning allows for smaller V-partitions. Smaller V-partitions' benefits cope with particular needs related to the NN (e.g., privacy concerns). It is possible to put optional split points between all the CNN layers, but some points are strategic since they allow to minimize the size of the feature maps transmitted, thus to minimize the overhead of the solution. These split points correspond to the outputs of the pooling layers. HyPS identifies all pooling layers as so called *Optional Split Points*. Algorithm 2 describes the identification steps of the strategic optional split points for a given CNN model.

Algorithm 2 Get strategic optional split points

Require:
$Model$: CNN model
$Threshold$: maximum number of parameters supported by the device
Ensure: LO : list of total strategic optional split points
for each $layer_i$ in $model$ **do**
    if number of parameters $<$ Threshold **and** layer type is Pooling layer **then**
        $LO \leftarrow$ index of $layer_i$
    else if the last layer **then**
        return LO

Algorithm 2 generates the list of all strategic optional split points. The benefits of using pooling layers include reducing the complexity and speeding up the calculations. Indeed, pooling layers are the most suitable output layer for the model partitions. This type of layers reduces the dimension of the output feature map resulting in a minimal data transfer between consecutive V-partitions. So, this work takes advantage of these layers to reduce the communication overhead of feature map transmission between V-partitions.

While splitting a CNN structure, there are two main scenarios. First, only a vertical partitioning is required because all V-partitions can then be executed without additional modifications onto targeted devices. Second, applying vertical partitioning solely does not sufficiently decrease partitions' complexity and an additional horizontal partitioning is required.

Apart from mandatory partition fit, the optional split points are helpful in several use cases when the user needs to distribute the partitioned model inference on multiple devices and wants to increase the number of V-partitions.

According to the user objective, HyPS provides final output partitions ready to be deployed on the target edge device(s) without any bottlenecks. HyPS can be integrated into a software program that provides an automated deploying solution of NN. Thanks to HyPS, the edge infrastructure can be covered more quickly. The model structure and edge infrastructure are passed as parameters. Three conditions are required to identify the strategic split points in the CNN structure:

- A large trained CNN model with chain topology,
- A well-defined resource-constrained device(s),
- Data for inference is available locally at the edge.

3.2.6 Application of Partitioning of VGG16

In this subsection, HyPS is applied to an example of CNN models, VGG16. This model is considered a huge NN. Therefore, we aim to split the structure of VGG16 into small partitions. These partitions can be executed at the edge without memory problems. HyPS allows the identification of the best split points to get the best partitioning result.

First, HyPS identifies the mandatory split points then generates the minimum number of partitions that allows the deployment of VGG16 on edge device. As mentioned in the Section 3.2.5, the study of CNN architecture reveals strategic locations through the CNN layers which are named optional split points. The optional split points allow to get a finer partitioning granularity.

In the case of VGG16 structure, there are five strategic split points (five pooling layers), so, there are several possible V-partitions. Figure 3.4 shows the VGG16 structure with the positions of the mandatory split locations represented by continuous red lines and optional split locations represented by dashed green lines. Mandatory split positions are located before and after the first fully connected layer(FC1). So the VGG16 must be partitioned vertically on this positions and split the FC1 horizontally to perform VGG16 inference on edge device. The optional split points bring the opportunity to get more than three V-partitions while minimizing the communication overhead.

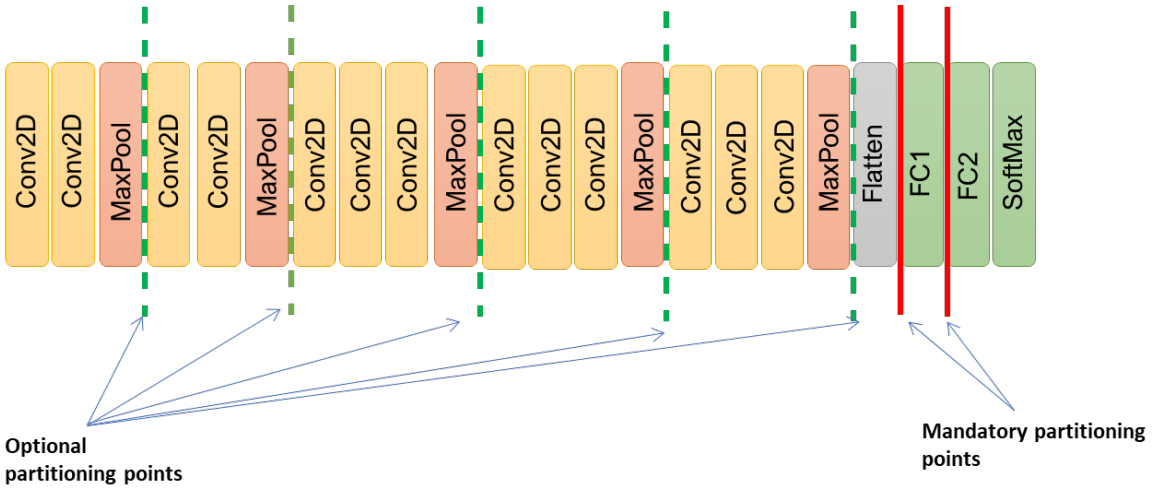

Figure 3.4: Mandatory and optional split positions of VGG16 model structure.

The pooling layers throughout the VGG16 model have different output dimensions, getting closer to the final output layer, the pooling layers output dimensions decreases. For example, in the VGG16 structure output feature map dimension passes from 112*112 in the

first pooling layer to 7*7 in the last pooling layer. Theoretically, it is more efficient to split the model on the last strategic point with the smallest feature map to minimize communication overhead.

The final result generated by HyPS comprised two elements. First, a list containing the mandatory split points as well as the optional split points. Second, the partitions that can be executed at the edge. By default, the partitions are generated according to the mandatory split points. Thus, HyPS split the VGG16 structure vertically into three V-partitions and the FC1 into four H-partitions. Four is the minimum number of H-partitions the target edge devices can support. The choice of the number of H-partitions will be detailed in the next chapter. It is possible to generate the partitions according to mandatory and optional split points to obtain smaller V-partitions. Table 3.2 presents the number of V-partitions according to the split points type in the case of the VGG16 structure.

Split points type	**V-partitions number (n)**
Mandatory split points	n= 3
Mandatory split points + optional split points	3 < n < 7

Table 3.2: Possible V-partitions number for VGG16 partitioning using HyPS.

3.2.7 Conclusion

The main objective of HyPS is to split the CNN structure into small partitions by mixing vertical and horizontal partitioning strategies while reducing communication overhead. This solution is different from the existing works by preserving the original structure without any modification, thus preserving the high accuracy and the model performance. The generated partitions can be deployed separately and distributed over time or spatially into multiple devices. It is required to define a scheduling schema to run the distributed inference process of partitioned CNN, ensure data transmission, and smooth execution of each partition at the edge. A scheduling policy will be presented in the following section to perform partitioned model inference on the edge infrastructure. An illustrative example is then described with a VGG16 CNN.

3.3 Architecture Overview and Qualitative Assessment of HyPS

This section comprises two parts. In part one, we define a specific architecture design that allows the distribution of model partitions at the edge and the scheduling of the execution of these partitions. This architecture will be used in the implementation afterward to ensure the computation of partitions and communications between different entities. In part two, we provide a qualitative assessment of HyPS via realistic use cases by identifying the benefits of adopting and applying our solution.

3.3.1 Distributing and Scheduling Architecture Overview

The architecture is composed of several entities that compose a multi-agents system. Two kinds of agents' preoccupations are mixed together and each of these kinds is structured as a tree topology. Each topology is in fact a view of HyPS.

3.3.1.1 Computations Topology

The computations topology comprises two types of entities: a *manager* and its multiple *workers*. The manager is responsible for the DNN model partitioning and the partition distribution across the available workers. Every manager manages the execution of a given DNN model through jobs by scheduling the execution of partitions on different workers. The different entities constituting the computations topology can be seen in Figure 3.5, the manager is denoted "M1" while the workers are denoted "W machine" where "machine" denotes the location of the worker execution. The manager assigns the job to its workers and decides the partitions taken by every worker.

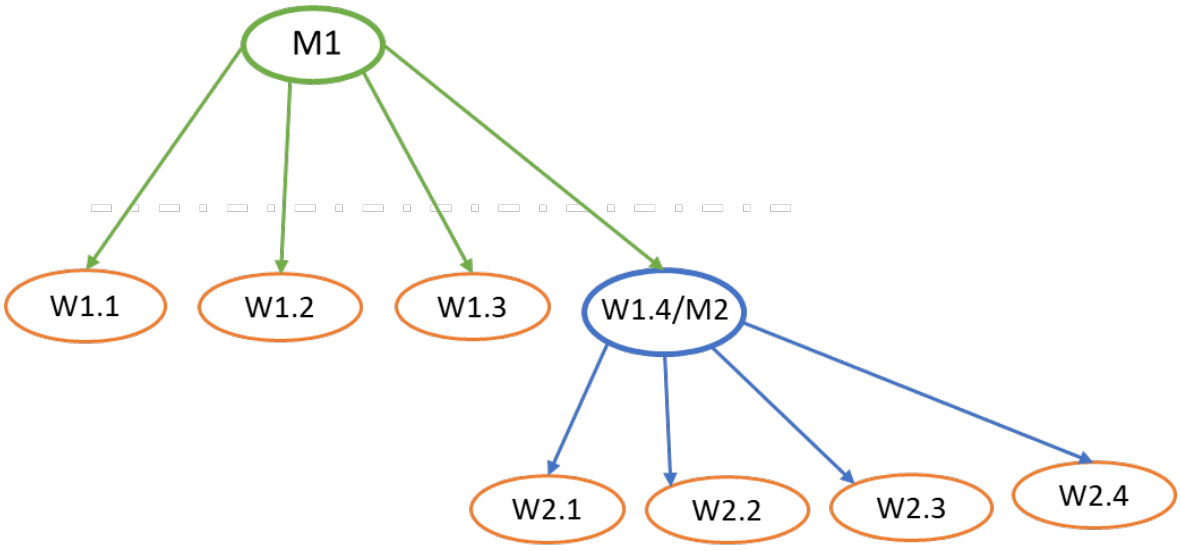

Figure 3.5: Architecture overview of computation topology

A manager can have workers that serve as sub-managers that manage a distinct group of workers. Two managers are represented in Figure 3.5 for ease of understanding. The manager M1 has four workers, and one of these workers, denoted "M2" is both a worker for M1 and a sub-manager of four other workers. Nonetheless, this number can be significantly higher in reality. A worker can receive and execute one or more partitions submitted by its direct manager. The green lines represent the computation links between Manager M1 and its workers, while the blue lines show the computation links between the second manager M2 and its workers. In Figure 3.5, two clusters are defined.

3.3.1.2 Communications Topology

The communications topology is an overlay of the computations topology. It manages data exchanges between entities. A communication hub transmits data between the manager and its workers. Both the manager and the workers know how to contact the communication hub that separates their communications thanks to a specific addressing policy. The communication hub allows for a reactive approach for each worker where each one do computations only when data are effectively present at a given address. In Figure 3.6, we present the communication view of the two clusters showed in the previous section (Figure 3.5). C1 and C2 are two communication hubs that ensure the data transmission within the corresponding clusters.

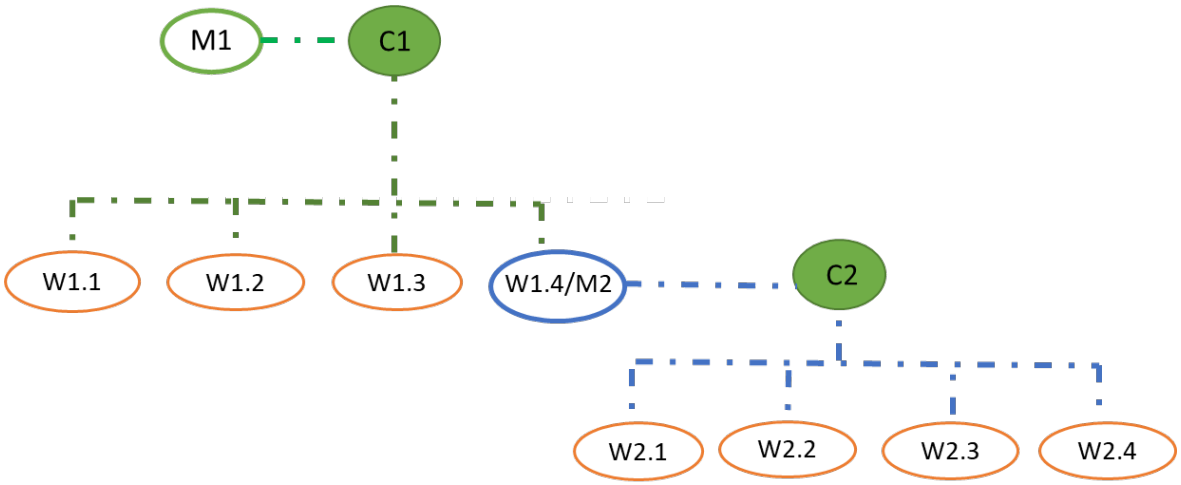

Figure 3.6: Architecture overview of communication topology

The communication hub is used to synchronize the tasks and the workers. The communication hub ensures that the workers can reach their manager directly. This hub is beneficial when there are sub-networks to manage. The communication allows the logging of exchanges and provides the history of executions carried out within a well-defined cluster. Thus, it is efficient for failure recovery from state saved thanks to the communication into a stable storage as presented in [113].

3.3.1.3 Scheduling and Execution of Model Partitions

The inference process using the HyPS architecture is composed of two main phases. The first phase consists in partitioning the given CNN model into partitions. The second phase involves the placement and scheduling of the partitions to obtain the final inference response. To provide better understanding, the HyPS system is represented as a Deterministic Finite Automaton (DFA). A DFA is a computational model that describes a limited number of possible states which can be reached during the computation process. For example, the life cycle of a given NN in the HyPS system can be represented by a DFA that includes different states and transitions from partitioning to computing. The functionality of a DFA can be described using a graph. Each state of the system corresponds to one vertex of the

graph. The edges represent the transitions between states. For each transition, a list of inputs that cause it are fixed. In DFAs, the transition from one state to another can happen only if the input matches the description of the transition.

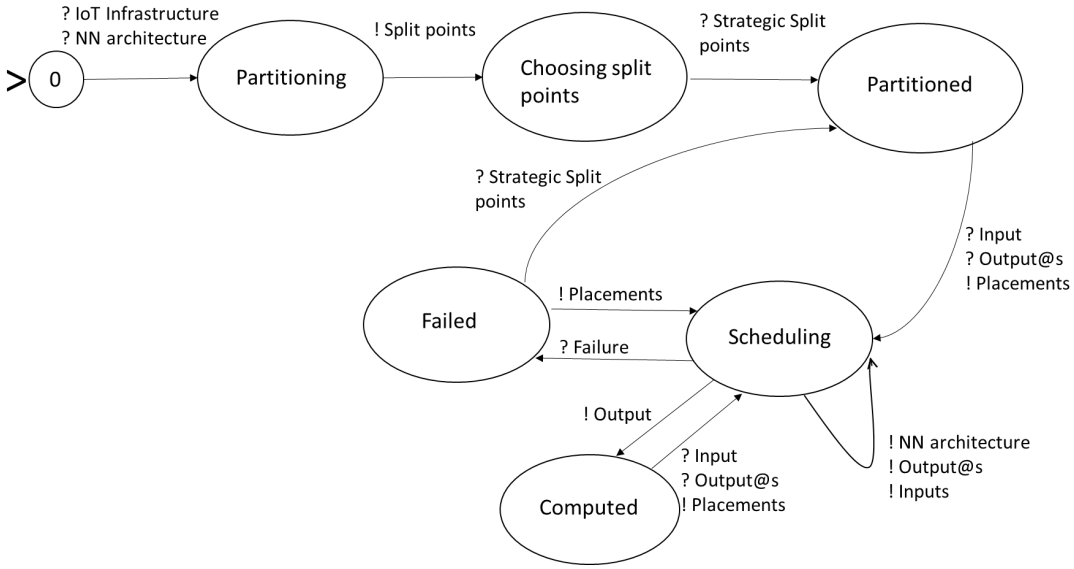

Figure 3.7: NN Inference state graph.

Figure 3.7 represents the DFA for a NN structure to be partitioned using HyPS. Transitions that start with the symbol "?" refers to input that is required to get the next state. Transitions that start with the symbol "!" refers to transition that produces an output that can be retrieved elsewhere as input. Some transitions require a list of elements given as input. For example, "Output@s" is a list of output addresses. When the execution of the HyPS system begins, it is required to get the NN architecture and the IoT infrastructure to move on to the next state " Partitioning". To move from state " Partitioned" to the state " Scheduling" it is required to get the input partition, the placement on a well defined worker and a list of addresses to send the output. Each worker receives the partition(s) and the address where to save the output feature map. The worker still waiting until the manager sends the input address that contains the appropriate input data.

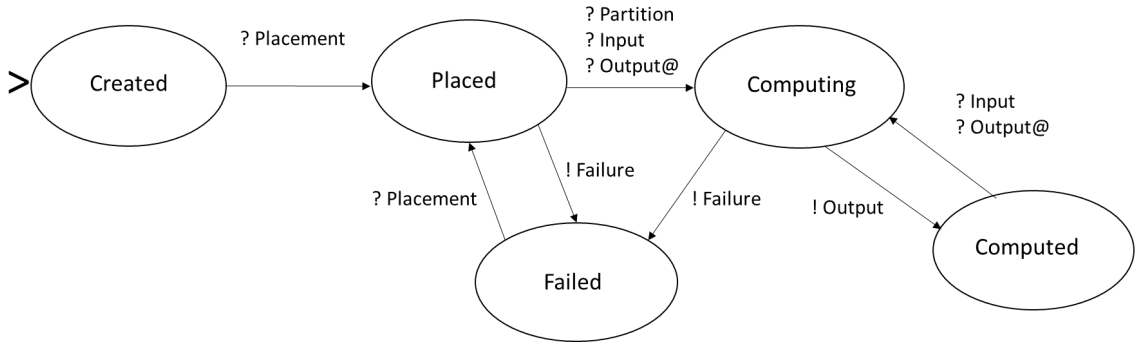

Figure 3.8: Partition life cycle graph.

The partition life cycle is described by a DFA that includes all the steps through which

passes the partition to be computed. Figure 3.8 represents a macro view of a partition inference and shows the different states from partition creation to the computation.

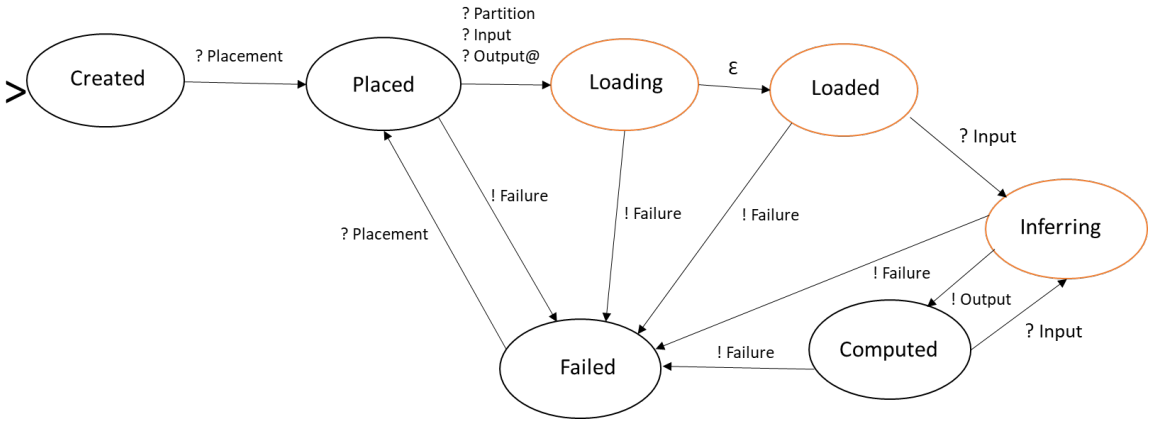

Figure 3.9: Partition life cycle detailed graph.

Figure 3.9 shows the partition's life cycle in details. The state " Computing" in Figure 3.8 includes three states " Loading", "Loaded" and "inferring" represented in orange circles. The manager is responsible for scheduling the partitions execution described in the algorithm 3. Before computing, one partition must be placed on a target device. The manager decides the placement of each partition. Once the partition is placed, the manager sends the input and output addresses, then the partition starts loading. When the partition is loaded, the manager sends the appropriate input data to start the computation. For the first partition, the input data is the original data that will be inferred. For the following partitions, the input data of a partition P_i is the intermediate feature map(s) generated by the previous partition P_{i-1}. The process is repeated until all available partitions are executed and the final output inference is computed.

The proposed architecture used in HyPS system gives the opportunity to run batch inference on a set of images. The Figure 3.9 highlights the advantage of performing batch inference using HyPS. These advantages include avoiding the waste of time related to the partition placement and loading for each image in the set. The algorithm 3 manages the batch inference process in which, the partition is placed and loaded only once, and the images will be computed one by one without repeating neither the placement nor the loading. When all the images are processed by the first partition, the manager authorized the next partition to start computing. The batch inference using HyPS architecture can be a good solution in several real-world use cases. For example, this approach can be used to process "cold data" when the computing load of the edge infrastructure is low.

In the following section, we describe the inference process on single and multiple devices of partitioned VGG16 using HyPS architecture.

Algorithm 3 Partitions scheduling

Require:

$listPartitions$: a list of partitions
$listInputData$: a list of input data
$InputData$: a peace of input data to compute
P: a partition
W: a worker
$OutputAddress$: an address contains the output feature maps
$InputAddress$: an address to get the input feature map
for each P_i in $listPartitions$ **do**
    if P_i is Placed **then**
        $W \leftarrow P_i, InputAddress_{(P_i)}, OutputAddress_{(P_i)}$ ▷ P_i is loading, computations can start
    else if P_i is loading **then** ▷ wait
    else P_i is loaded
        if i=0 **then**
            $list \leftarrow listInputData$
        else
            $list \leftarrow$ get $OutputAddress_{(P_{i-1})}$
    for each $InputData_i$ in $list$ **do**
        $InputAddress_{(P_i)} \leftarrow InputData_i$ ▷ P_i is inferring ▷ wait for output

3.3.2 Inference on Single Device

Partitioned CNN inference can be performed on one device providing that the partitions execution is distributed over time. The manager send all partitions to a single device however the computation workflow is ran sequentially. Figure 3.10 depicts a possible scheduling scenario for partitioned VGG16 using the proposed hybrid approach. In this case, the VGG16 is partitioned into four V-partitions denoted Vp1, Vp2, Vp3 and Vp4, represented by blue rectangles. The first fully connected layer (L19) is divided into four H-partitions: Hp1, Hp2, Hp3, and Hp4, represented by green rectangles. Four is the minimum number of H-partitions for which the number of parameters is sufficiently reduced to fit a typical edge device such as a Raspberry Pi 3B+. The edge device computes inference calculations in a serial first-in first-out manner. Tnumber represents the consecutive partitions' instantiation and execution time. For example, V-partition 3 (Vp3) in Figure3.10 is executed only when feature map coming from computations of Vp2 is finished and received by the communication hub. Instantaneous memory footprint when running the inference process on a single device is lower than memory consumed when running the whole model in one block.

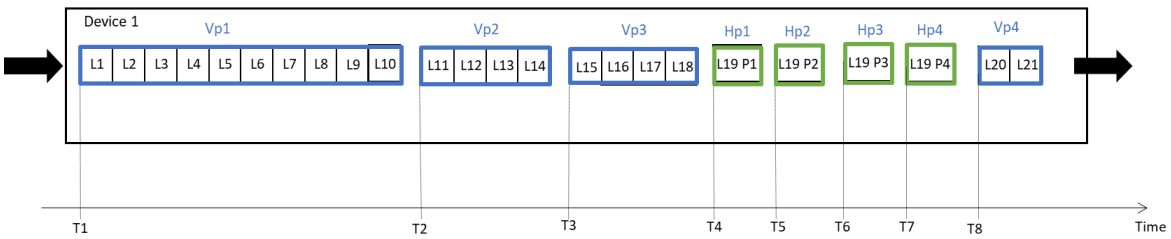

Figure 3.10: Partitioned VGG16 inference on single edge device

HyPS allows executing a large CNN model on a single edge device successfully even though this model was not running before on a resource-constrained device. This result is an advanced contribution of HyPS. The performance measures exposed later in this manuscript show in some cases an improvement in the execution speed.

Performing partitioned CNN inference on a single device is beneficial when the data can only be processed in a single place. For example, it may be the case for a user's home equipped with a single device(e.g., an internet gateway). The edge infrastructure makes it possible to broaden the inference at the edge by federating a set of resource-constrained devices which will be described in the following subsection.

3.3.3 Distributed Inference on Multiple Devices

Inference can be distributed over time and spatially across multiple edge devices without any cloud processing. Figure 3.11 shows an example of a scheduling for distributed inference of a partitioned VGG16. All partitions are distributed across four edge devices. Partitions' execution order is determined according to initial position of each partition in the initial architecture. V-partitions are always executed sequentially, only the H-partitions of the FC1 layer can be performed in parallel as shown in Figure 3.11. The results of H-

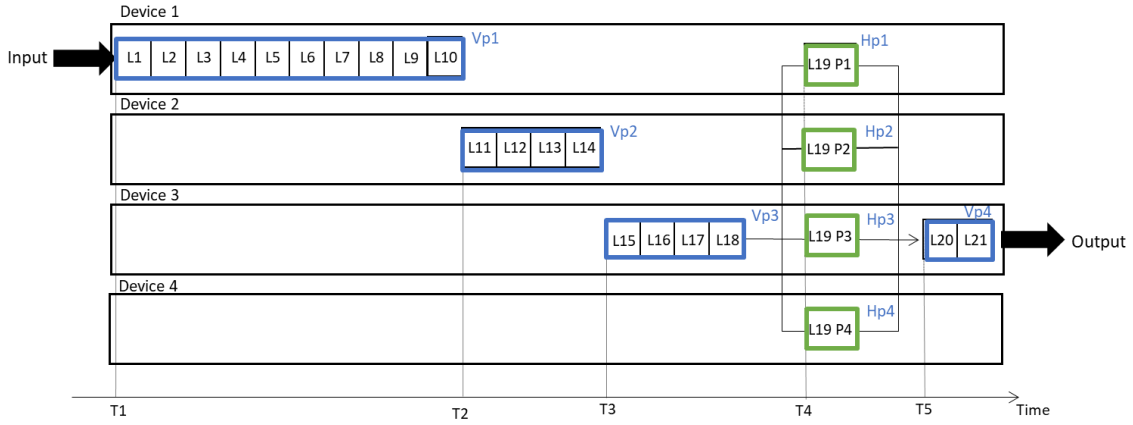

Figure 3.11: Hybrid partitioning of VGG16 model deployed on a cluster of four edge devices.

partitions of the same layer must be assembled together before sending the feature map to the communication hub.

Distributing the inference process across multiple devices is a good way to perform a large CNN inference at the edge, but there are other reasons to exploit multiple devices. The distributed infrastructure is a better track to execute the inference on a batch of existing observations or observations that can be generated in real-time. Performing the computations separately on edge devices allows for the acceleration of the inference process. It avoids the disruption of the main function of the device(s)(e.g., connectivity, recording video, etc.)

Figure 3.12 shows a prototype of distributed inference of partitioned VGG16 that depicts an industrial use case inspired by [114]. The manager denoted "M1" is the manager of four workers, W1, W2, W3, and W4. Regarding knowledge of model and data, entities are categorized into two classes. The manager and the communication hub are classified as safe entities because they must be deployed on a device with robust protection policy. This high protection is necessary to protect all information about the original NN structure. The workers are classified as unsafe, and there is no need for high protection because workers have only a part of the NN structure.

The proposed architecture, scheduling policy, and the different entities categorization allows considering HyPS as a better approach for performing inference at the edge. HyPS may have an efficient impact in many real-world cases. In the following paragraph, we highlight the main improvements that can be gathered by applying HyPS to examples of real-world IoT applications.

3.3.4 Qualitative Assessment of HyPS via Concrete Use Cases

The deployment of current AI applications at the edge suffers from numerous limitations in failure resistance, data confidentiality, service efficiency, and scalability issues. HyPS proposes solutions to these limitations while guaranteeing high accuracy and lower risks. This section presents three realistic applications of our proposed strategy: 1) AI-powered Security Camera, 2) Smart Manufacturing, and 3) Autonomous Driving. Through these use

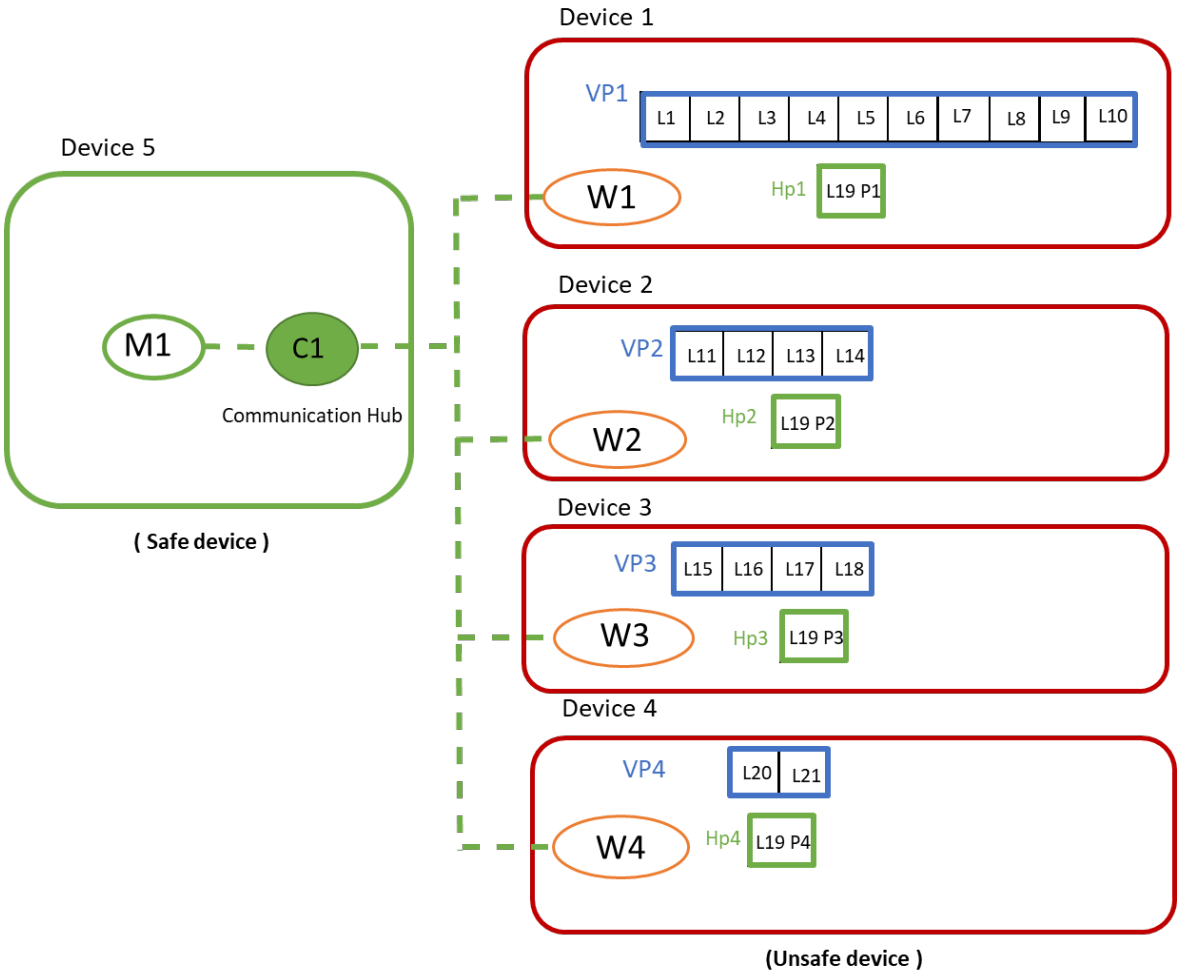

Figure 3.12: Example of distributed inference architecture of partitioned VGG16 model on a cluster of four edge devices.

cases, we highlight the improvement brought by HyPS.

3.3.4.1 Use case 1: AI-powered Security Camera

Image recognition, also called image classification, is an important task in the computer vision field. Image recognition is used to identify certain types, aka classes, of objects within an image or video frame. Image recognition can be carried out through simple image processing methods. However, these techniques can be quite restrictive in functionality and do not provide high accuracy. The integration of CNNs into image recognition applications improves results but image processing becomes more resource-consuming. So, executing this type of application, including large AI models near IoT devices, is challenging at the edge. HyPS can solve this problem. For example, a user wants to use a software service for image recognition linked to a connected security camera to perform advanced tasks such as person detection, vehicle detection, and traffic counting.

This service can be integrated into any embedded device. Users want to integrate this application simply on the available infrastructure. Internet gateway (a.k.a. Service Provider (ISP), a nearby micro data center) can be used to perform image recognition collected by the camera. HyPS allows the partitioning and the deployment of a large trained AI model on multiple nearly connected devices which will help the user to analyse data and perform image recognition with high accuracy.

However, devices can be abruptly unplugged, so all the processed data can be lost, and the user must restart the inference again. The participating devices exchange data and intermediate inference results via the communication hub. So, in case of device failure, HyPS avoids restarting calculations from scratch, and permits to resume inference from the blocked point. So, The first improvement of using HyPS is :

- ❖ Resilience & Reliability.

In addition to partitions' computations scheduling, HyPS presents different advantages concerning edge infrastructure specificities, in particular resilience issues. If IoT devices and edge devices are numerous, they have limited computing and memory resources but also they can suffer from many dysfunctions. Among them, energy cutoffs can occur as it is mentioned in the use case above. Customers often unplug abruptly the gateway to save energy. HyPS offers resilience to NN inference execution at the edge by enabling backups policies such as those described and used for the IoT in [113]. HyPS permits the recovery of already done computations because the communication hub allows the storage of the intermediate computation results exchanged between workers. Such a reliable storage is fundamental in case of a problem during the inference process as the stored data can then be used as backups to regain the inference without restarting from scratch. Furthermore, thanks to distributed infrastructure, it is possible to run the same partition on several edge devices. This redundancy guarantees a high quality of service for the user and reduces the response latency in case of failure. It allows to compare inference results of the same partition on multiple computing nodes and carry out controls or voting systems to avoid erroneous calculations.

3.3.4.2 Use case 2: Smart Manufacturing

A second use case is a digitized manufacturing facility that uses connected devices, machinery, and production systems to collect and share data continuously. The smart factory aims to provide high-quality production lines through AI's intelligent decision and low latency edge computing processing. An intelligent robot is designed and developed to rapidly control productivity and detect defective pieces. This robot uses image recognition models. This smart factory is composed of different production areas. Each one takes charge of a part in the manufacturing process. The available machines in the production areas are very constrained in memory and computing capacity. So, running the whole AI model on one machine can increase the computational load, cause memory saturation, and increase subsequent response latency.

Also, these machines must give a real-time response. Training an AI model for a manufacturer is very costly in terms of time and computing resources, so applying a high-performance trained model is preferable. Generally, the manufacturer wants to exploit the machines available in the factory and avoids buying powerful specific machines for image recognition tasks nor transmitting his private data to cloud servers. The manufacturer priority is to prevent data disclosure and respect strict privacy protocols. The manufacturer may choose between evolving his production process using AI technologies or respecting the data confidentiality protocols. HyPS provides a well suited solution for this situation by running a partitioned model on one or multiple machines. The collected data is processed locally and is not accessible by other production units. Only intermediate data will be transmitted between different machines. So, our proposed strategy helps the factory to improve the quality of construction pieces without slowing the production process or causing damage to machinery. It also guarantees data and model confidentiality since each machine only deals with specific partitions of the global model. So, other improvements of HyPS are about :

❖ Scalability

HyPS provides a well suited solution for the manufacturer by sequentially running a partitioned model on single machine. Indeed, the execution of one partition will be less expensive than the whole model in one block. HyPS allows large model inference on a single machine, which is impossible without hybrid partitioning strategy. Thus, a machine with limited memory and computing capacity can perform a large model inference. So, the proposed solution allows processing large models on the resource-constrained machine(s). That's why it is a scalable solution for inference at the edge.

❖ Security policies compliance

The designed architecture of HyPS is related to security and, more precisely, all possible information disclosures. Possible disclosures comprise the data passed in input and obtained from the output of a given NN and the NN architecture itself. If data can unveil industrial or private secrets, NN models are assets in which investments were made for their design and training. To the best of our knowledge, the literature weakly covers this second aspect. Preserving both of them is an important concern that HyPS addresses.

Going back to the detail of the proposed prototype, each worker only knows its manager, the communication hub, a part of the entire model, and the data it processes. Also, the manager knows all the architecture, all the workers, the communication hub, and the data processed. The communication hub has the same knowledge as the manager. As a consequence, two kinds of entities can be identified with privacy concerns as an objective. The manager and the communication hub must be under high security. They must be hosted on safe devices that cope with firm security policy (represented in the green rounded corner rectangle in Figure 3.12). The workers can be deployed on "unsafe" devices (described in the red rounded cornered rectangles in Figure 3.12). Regarding entities' knowledge about described secrets in the Figure 3.12 example

- Only M1 and C1 know the full AI model,
- Only M1 and C1 know the entire data,
- Other entities only know a part of the AI model,
- Other entities only know a part of the data.

Thanks to these different roles, **the manager can not divulge data and the NN structure**. Workers outside the safe zone will receive only partial data already executed. HyPS uses the distributed inference process to guarantee data privacy even when workers run on unsafe machines. Indeed, only the manager and the communication hub have the actual image, while related workers (including sub-managers) only have intermediate feature maps.

In the absolute, one manager is not aware if it knows the entire NN and the actual data because it could be a sub-manager. Therefore, revealing the original image after layers carry out many operations in each partition is difficult. It is demanding to interpret the partial data since having undergone several unknown transformations. This proposed execution process allows to take advantage of the presented infrastructure's design by protecting data, including workers with weak security policies.

3.3.4.3 Use case 3: Autonomous Driving

An autonomous vehicle is a car that can operate itself and controls all aspects of driving without any human intervention. These vehicles are embedded with onboard sensors and AI models that allow sensing the environment. In addition, autonomous cars need to make real-time decisions. A high latency in response time will result in severe consequences. Running AI models locally and moving computing tasks to the network's edge (e.g., vehicles) is an efficient solution to reduce latency and avoid data transmission delay. For example, HydraOne [115], and HydraMini [116] are typical examples of autonomous vehicles that are equipped with embedded computing platforms to support AI (e.g., CNN) inference and traditional computer vision analysis and can make real-time decisions.

For autonomous cars, computer vision applications need to recognize numerous objects and obstacles on the road. Inference in a batch of observations collected in real-time is required to detect objects quickly. HyPS provides an efficient way to perform inference in batch using large CNNs on autonomous cars. Deploying a partitioned CNN on this type of car allows for reducing data processing delay. Processing the data collected by the car sensors locally allows a significant gain in response time with high precision. Furthermore, avoiding data transmission to the cloud enables the protection of private information related to the driver's location, journey, and final destination. Another improvement of HyPS is about:

- ❖ Efficiency

HyPS architecture and scheduling policy allow to run large CNN inference on single device. The execution of the model partitions is distributed over time and partitions are inferred one by one. So, the instantaneous memory occupied by a partition during computation is reduced. The target device takes charge of one partition of the whole model at a time. Once a partition is inferred, the worker loads the following partition. Performing the inference process on a single worker brings an opportunity to launch the inference process of a voluminous model without requiring cloud computing capacities. Most of autonomous cars are embedded with at least one edge device. HyPS performs object detection tasks instantly and locally on edge devices.

3.4 Conclusion

HyPS provides the best way to partition a large CNN structure while reducing latency and communication overhead. Converting a big model to small partitions allows the industry to exploit several existing trained AI models in many real-world cases. This contribution is an efficient way to use trained AI models on edge devices while providing many improvements. Furthermore, HyPS allows high-precision prediction without moving the data from its device. The next step is evaluating the proposed strategy on a real test bed and providing experimental results of distributed inference of partitioned CNN at the edge.

Chapter 4

Implementation and Evaluation of Proposed Hybrid Partitioning for CNNs Inference at the Edge

4.1 Introduction

In chapter 3, we have presented the main contributions of this thesis describing the hybrid partitioning strategy and the designed architecture for implementation. This chapter is devoted to the implementation and evaluation of the proposed hybrid partitioning approach applied to the governing example used in this manuscript, VGG16. This chapter includes two sections. First, section 4.2 details the implementation of HyPS on a real edge infrastructure. Second, in section 4.3, we evaluate the performance of the proposed solution compared to existing approaches.

4.2 Experimental Set-up

4.2.1 Test bed Description

In order to evaluate the performance of proposed hybrid partitioning strategy on an edge infrastructure, we chose the Raspberry Pi 3 Model B + hardware platform. It is based on a Broadcom BCM2837 System on a Chip (SoC) including a Quad Core ARM Cortex-A53 1.2GHz 64-bit CPU as well as a Broadcom VideoCore IV GPU. It also features 1GB RAM LPDDR2 900MHz and a storage capacity depending on the microSD card manually inserted. The ARM processor works at frequencies ranging from 700 MHz to 1.2 GHz [117]. The testbed includes three Raspberry Pis 3 B+ and a PC with a Linux operating system. The PC is used only in certain cases for which a partition can not be run on a Raspberry Pi. This particular case will be described further. Experiments are done using the VGG16 NN model that is specified by a chain topology and trained onto the ImageNet dataset. In the following experiments, images of fixed size of 224x224 are used.

4.2.2 Software Architecture

To establish the communication between Raspberry Pis, Eclipse Mosquitto v1.6.2 has been used, one of the most known MQTT servers [118]. Eclipse Mosquitto is used as an MQTT broker to ensure the manager's and workers' addressing process. The default maximum number of possible connections is around 1024 [119]. To facilitate the application of the hybrid partitioning strategy and launch the inference process easily, we use FastAPI, a modern web framework for building APIs with Python 3.7+ based on standard Python type hints. Swagger UI allows to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from the OpenAPI (formerly known as Swagger) specification, with the visual documentation making it easy for back end implementation and client side consumption. Figure 4.1 shows an example of the web interface that allows to apply the hybrid partitioning strategy with one click.

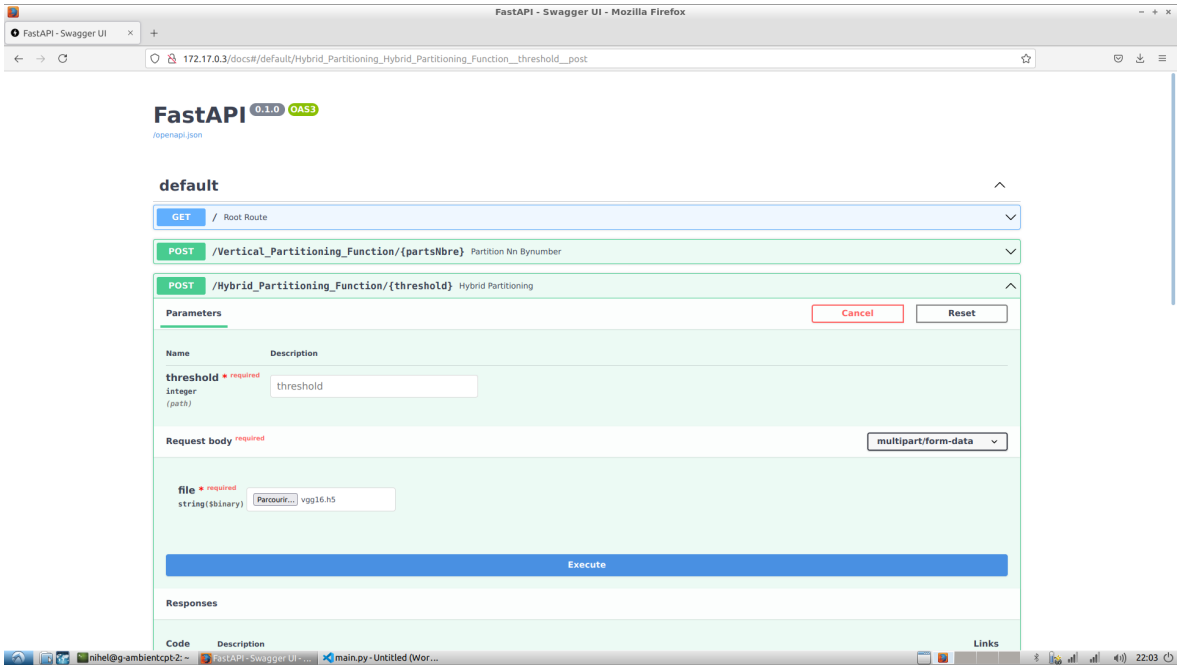

Figure 4.1: The manager's web client interface

4.2.3 Implementation

HyPS' workflow is composed of three steps, Model Partitioning, Partitions Assignment, and Distributed Inference & scheduling step. The final result of applying HyPS provides the generation of partitions, their deployment, and their distribution on edge devices.

4.2.3.1 Model partitioning

The Hybrid partitioning function was implemented in Python using TensorFlow[120], and Keras [121]. The proposed function takes as input the original model to be partitioned and the maximum number of parameters supported by the target device. Experiments show that the maximum number of parameters only concerns the dense layers. This number can be easily determined by a test and fail method, consisting in testing different partitions sizes. The limit value of the number of parameters depends on the hardware characteristics of the device. The output is the model partitions according to the mandatory and the optional split points if they exist. The Hybrid partitioning function is applied to produce new Keras models with the desired layers. Each partition is consequently a standalone Keras model that can be deployed without further modification. These Keras models can be executed separately without modifying the layers' order or the weights. The implemented function specifies whether it is a vertical or horizontal partition. The Hybrid partitioning function has been implemented using Keras's core original function: using HyPS, the user can thus run our solution easily.

4.2.3.2 Partitions Assignment

According to the architecture designed in section 3.3.1 in chapter 3, the manager is responsible for the partitioning step and assignment of partitions to workers. In the proposed testbed, this manager is run on the PC to represent a node present outside of client homes, inside a typical telecommunication infrastructure while Raspberry Pis represent Internet Service Provider (ISP) gateways. The manager is responsible to distribute the partitions across the available workers which are the Raspberry Pis. It is worth noting that the manager can be run on a Raspberry Pi.

In the implemented prototype, the Communication Hub is a Message Queuing Telemetry Transport (MQTT) broker, and more precisely, a Mosquitto [118] instance. Communications between all entities are managed through this broker. It is a straightforward and lightweight publish/subscribe based messaging protocol for constrained devices and networks with high latency, low bandwidth, or unreliable networks. MQTT has been used in this implementation as a communication support, but other protocols could have been used such as Advanced Message Queuing Protocol (AMQP), or connected ones (e.g. websockets). The protocol's design principles are to minimize network bandwidths and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery [122].

In a publish/subscribe (pub/sub) communication model, components interested in consuming certain information register their interest. This process of registering an interest is called subscription, the interested party is therefore called a subscriber. Components which want to produce certain information do so by publishing their information. They are thus called publishers. The entity which ensures that the data are transmitted from the publishers to the subscribers is the broker. The broker coordinates subscriptions, and subscribers usually have to contact the broker explicitly to subscribe [123].

There are three principal types of pub/sub systems: topic-based, type-based and content-based [124]. MQTT is a topic-based system, the list of topics is usually known in advance. The manager define all the topic according to the partitions assignment policy. A strong advantage of MQTT over a websocket approach is the ease to add features regarding restart after a failure as each message can be easily traced, duplicated for backup and kept in queue. Moreover, MQTT is widely used in Orange services.

4.2.3.3 Distributed Inference & Scheduling

After the partitions assignment step, the manager sends data to the MQTT broker. The addressing policy of MQTT allows task scheduling. Each worker waits for its tasks at a specific address and returns its calculations in an agreed topic. MQTT supports basic end-to-end Quality of Service (QoS) [125]. Depending on how reliably data should be delivered to the workers, MQTT distinguishes between three QoS levels. QoS level 0 means that the message is sent once and that delivery is not guaranteed. QoS level 1 provides a more reliable transport: messages with QoS level 1 are re-transmitted until they are acknowledged by the receivers. Consequently, QoS level 1 messages are certain to arrive at least once but

may arrive multiple times at the destination because of the re-transmissions. The highest QoS level, QoS level 2, ensures not only the messages' reception but also that they are delivered only once to the receiving entities. This parameter avoids data confusion between workers. In the implementation, we chose the QoS equal to two for sending partitions. So, we guarantee that each worker receives exactly once the appropriate partition.

Upon receiving data on the incoming topic, the worker performs the needed operations and runs inference through its partition. Once the output is computed, the worker serializes data to be sent to the MQTT broker. These steps are repeated for each compute node. The manager receives the final inference response, the output of the entire NN if no partitioning has been done when all workers finish their jobs.

4.3 Evaluation of the proposed Hybrid Partitioning Approach

This section presents the overall experimentations carried out to evaluate HyPS and assess its efficiency. It contains analysis of experimentation's results and a validation of the improvements mentioned in chapter 3. In the next subsections, the following metrics are measured for each experiment:

- *Inference time:* the time necessary for the whole inference. It includes both the computation and the communication time.
- *Computation time:* the time required to perform a computational process of inference per partition.
- *Communication time:* the transfer time of intermediate feature maps from one partition to another.

In the following subsections we propose to evaluate in section 4.3.1 the impact of the vertical and horizontal partitioning strategies before studying the impact of hybrid partitioning in section 4.3.2.

4.3.1 Impact of Vertical and Horizontal Partitioning

Experiments are performed in two steps. The first step consists in the identification of the mandatory and the best optional split points in the model structure. For this purpose, the vertical partitioning is applied in different locations on the VGG16 structure to compare the *quality of partitioning* for each case. The quality of partitioning refers to the validation of final inference output when using a well-defined partitioning strategy, according to specific criteria cited as follows:

- obtaining final inference output : in the case of VGG16, the detection of the object given in the input image.
- maintaining high accuracy and performance : giving the suitable class of the detected object.
- obtaining a reasonable inference response time compared to the device's capacity.
- reducing the communication overhead.

After choosing the best positions to split the model, the horizontal partitioning is applied in the second step to enable complex layers execution on the edge device. This second step is discussed further noticeably to measure the impact of the Hybrid partitioning proposed in HyPS on the inference performances.

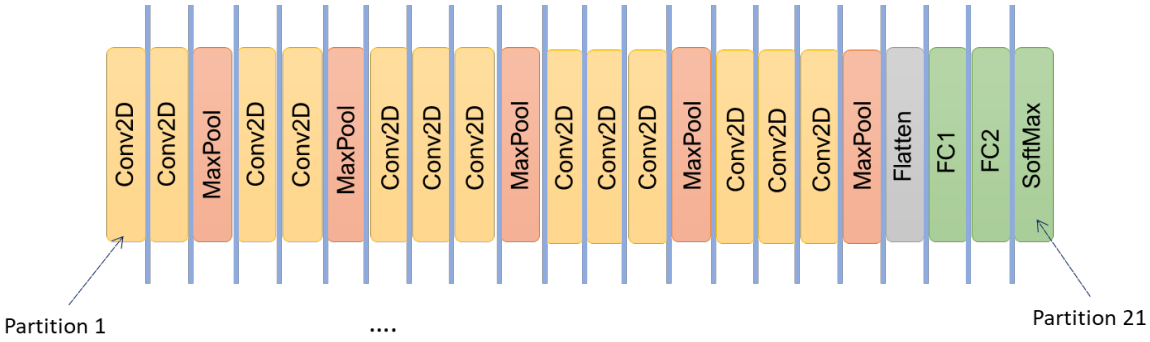

Figure 4.2: VGG16 model partitioned vertically on 21 partitions.

The first experimentation objective is to evaluate the impact of vertical partitioning on communication overhead and inference time. The VGG16 model is partitioned vertically to get the smallest possible V-partitions: one partition includes only one layer. Such a partitioning can be refined, but it is a suggested way to pre-qualified a NN before execution. Indeed, if a partition cannot be run, it must be partitioned horizontally. For this pre-qualification round, 21 V-partitions are generated since the model contains 21 successive layers. Figure 4.2 shows the positions of vertical partitioning on VGG16 model.

After partitioning, the obtained partitions are executed on a single Raspberry Pi. An error has occurred in the V-partition that contains the first fully connected layer. Because this layer is too complex, it cannot be executed without horizontal partitioning. Fixing this particular complex layer allows specifying the first mandatory split points in the VGG16 structure.

In practice, only the first fully connected layer(FC1) causes the problem; all the rest of the layers run on Raspberry Pi. So, this experiment allows defining the mandatory split and fixing the threshold supported by the available device. This particular complex layer is discussed further, but the problematic V-partition is simply offloaded onto a PC for the current experiment. In this experimentation, the measures of the inference time of the FC1 on the PC is ignored as it is not comparable with others. Only measurements of the V-partitions executed on edge devices are under consideration.

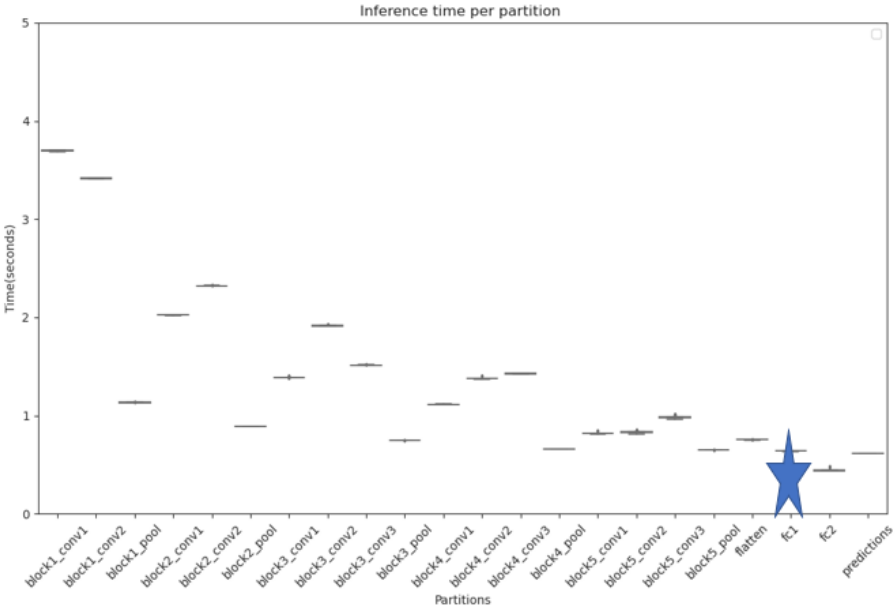

Figure 4.3: Inference time of partitioned VGG16 model vertically on 21 partitions deployed on Raspberry Pi.

Figure 4.3 shows the inference time for the 20 V-partitions runnable on a Raspberry Pi. From the input to the output layer, the inference time overhead decreases on the whole. Some local minima appear specifically with partitions that contains pooling layers.

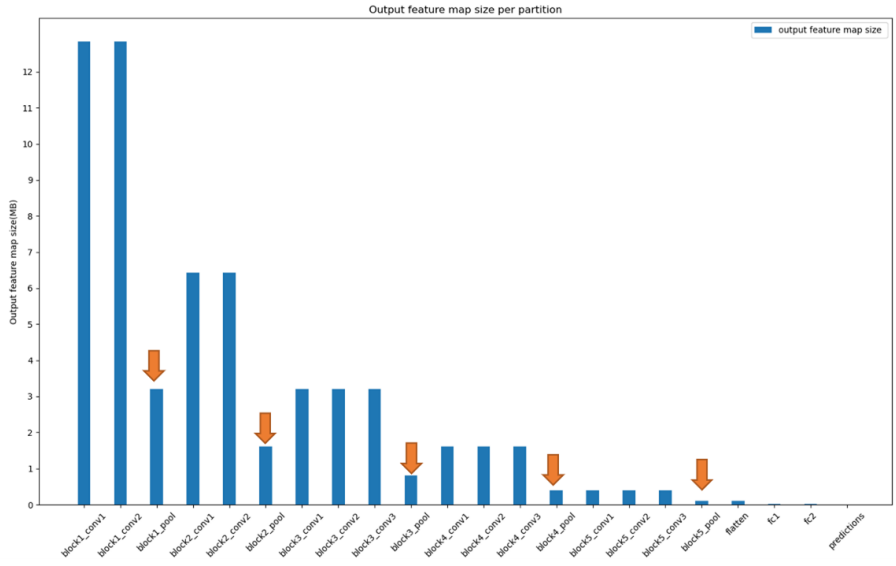

Figure 4.4: Output feature map size per V-partition.

Figure 4.4 depicts the feature maps size generated by successive partitions. This decrease is explained by the reduction of the dimension of the feature maps generated by the pooling layers. The size of data transmitted between V-partitions appears to be directly correlated to the feature map dimensionality. Chart in Figure 4.4 shows local minima in

the feature map size map generated by the pooling layers that correspond to local minima regarding the inference time of the VGG16 partitions depicted in Figure 4.3. Orange color arrows indicate these local minima in Figure 4.4, which correspond to pooling layers in the VGG16 structure.

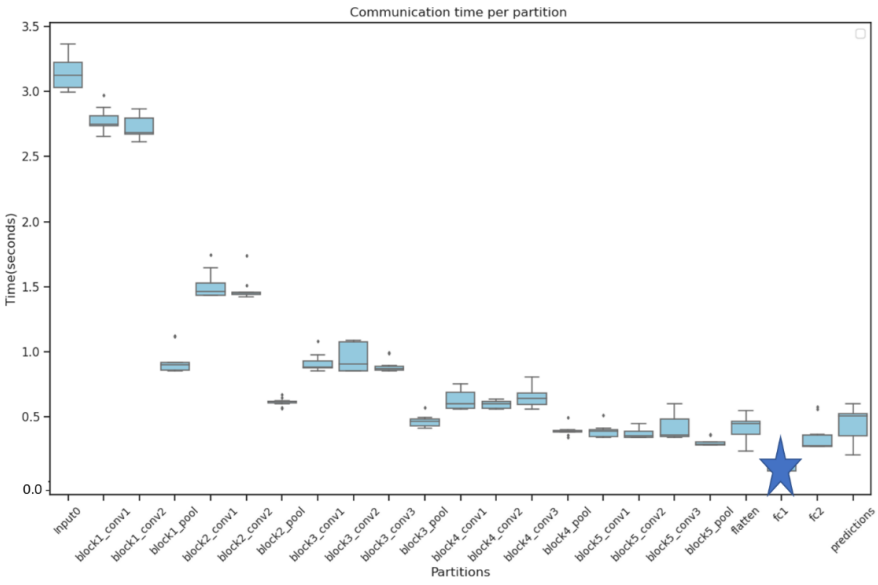

Figure 4.5: Communication overhead of partitioned VGG16 model vertically on 21 partitions deployed on Raspberry Pi.

Figure 4.5 presents the communication overhead for the VGG16 V-partitions. The communication overhead depicts 70% of the overall inference time. It appears that the communication overhead and the inference time depend on the dimension of the feature map, the larger the feature map shape, the slower the transmission speed of the feature map between layers. To minimize the communication overhead, splitting the model after the pooling layers is efficient. These positions represented the optional strategic split points. The number of parameters does not impact the communication overhead because the convolution layers with the lowest communication overhead are the layers with a high number of parameters.

Browsing the model structure, all the pooling layers do not have the same impact on the communication overhead. The pooling layers close to the model's output layer permit minimizing communication overhead more than pooling layers close to the input layer. Therefore, the optional split points close to the output layer should be favored.

The following experiment aims to validate the quality of vertical partitioning on pooling layers compared to the other layers in the model structure. We propose comparing the communication overhead for partitioning on pooling and convolution layers. Figure 4.6 shows a vertical partitioning on two locations in the VGG16 structure. Figure denoted (a) presents the VGG16 model partitioned into nine partitions; the output layer of the first five partitions is a convolutional layer. Figure (b) gives the VGG16 model partitioned on nine partitions; the output layer of the first five partitions is a pooling layer.

The main goal of this experiment is to measure the communication overhead in the

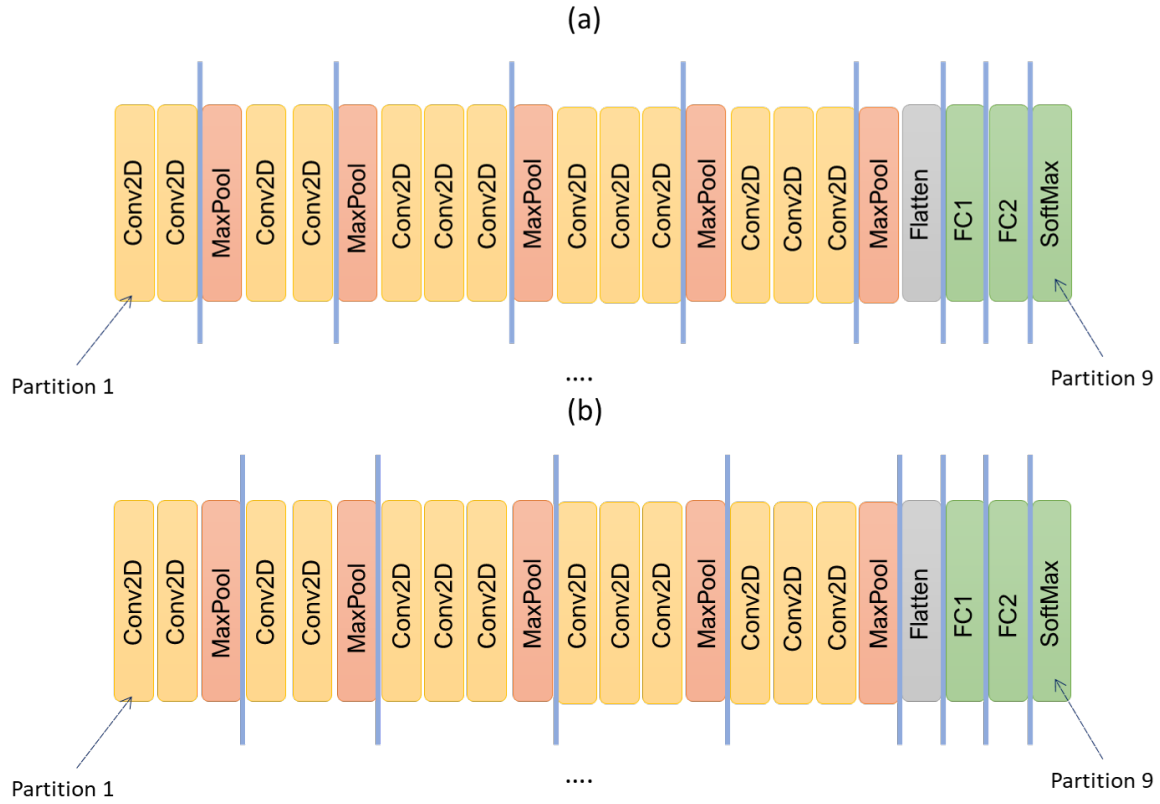

Figure 4.6: (a) VGG16 model partitioned vertically on convolutional layers (b) VGG16 model partitioned vertically on pooling layers.

two cases and compare the results. The convolutional and pooling layers have a different roles. The convolutional layer serves to detect patterns in multiple sub-regions in the input feature map using different filters. In contrast, the pooling layer progressively reduces the representation's spatial size, reducing the amount of computations in the CNN. In the two ways of partitioning, the transmitted feature map's size differs because the output layer in the nine partitions is not the same.

The box plots (a) and (b) in Figure 4.7 show, respectively, the overall communication time when the output layer in the partitions of the VGG16 are convolutional layers and pooling layers and deployed on Raspberry Pi 3B. The values measured for the FC1 are not counted in the two graphs (a) and (b) because it is offloaded on a PC. For the first five partitions, the communication overhead when the output layer is a pooling layer is lower than the communication overhead when the output layer is a convolution layer. For the partition n°1 in the graph (a), the communication time is two times higher than in graph (b). The difference between the cases is the size of data transmitted between partitions because the pooling layers reduce the shape of the feature map generated by the convolution layer just before. Graph (a) shows a high standard deviation for the partition 6 which can be intolerable when processing data in real-time.

To conclude, these experiments show that:

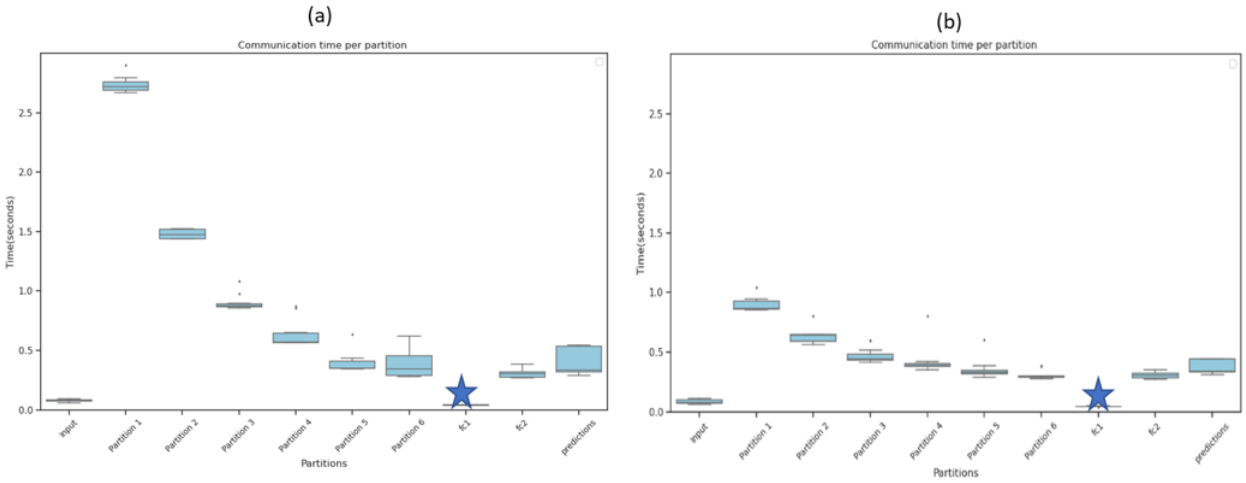

Figure 4.7: (a) Communication time of partitioned VGG16 on convolutional layers (b) Communication time of partitioned VGG16 on pooling layers.

1. **the best optional positions to split the model and reduce communication overhead are the pooling layers**,
2. **applying only vertical partitioning is not enough to deliver VGG16 inference on an edge device with limited resources**,
3. the V-partition that obstructs the inference process contains the first fully connected layer(FC1). That's why two mandatory split points are required: after and before the FC1 to isolate this particular layer into a separated V-partition that will be then H-partitioned. **This demonstrates the relevance of the proposed hybrid partitioning strategy**.

The previous experiments shows that vertical partitioning is insufficient to run partitioned VGG16 and deliver inference output response on single edge device. Therefore, it is required to apply horizontal on the complex V-partition to get smaller partitions that can fit to the edge device. In the following section, Hybrid partitioning strategy is applied on VGG16 structure to get efficient partitions that fit to Raspberry Pi without offloading on PC. After partitioning, VGG16 inference is performed sequentially on a single Raspberry Pi and spatially into three Raspberry Pis.

4.3.2 Impact of Hybrid Partitioning

4.3.2.1 CNN Inference on Single Device

HyPS is applied on VGG16 model to perform inference on single Raspberry Pi. VGG16 structure is partitioned first vertically on the mandatory split points. These points are shown in red lines in 3.4 in section 3.2.5 then the complex layer FC1 is partitioned horizontally into H-partitions. VGG16 partitions' execution is scheduled sequentially on the same Raspberry Pi.

First tests were done using numbers of H-partitions that cover a wide range. During the inference executions, it appears that a too small number of H-partitions leads Raspberry Pis to swap, resulting in very poor performances due to Raspberry Pis ressources depletion. The swap activation adds more virtual memory, allowing the system to deal with more memory-demanding tasks without out-of-memory errors or having to shut down other processes. However, the downside is that accessing the swap file significantly causes excessive energy consumption, slows down the process, and finally increases the inference time. This experimentation aims to identify the minimal number of V-partitions and study the impact of the number of V-partitions on the communication overhead. As a reminder, the swap had been disabled.

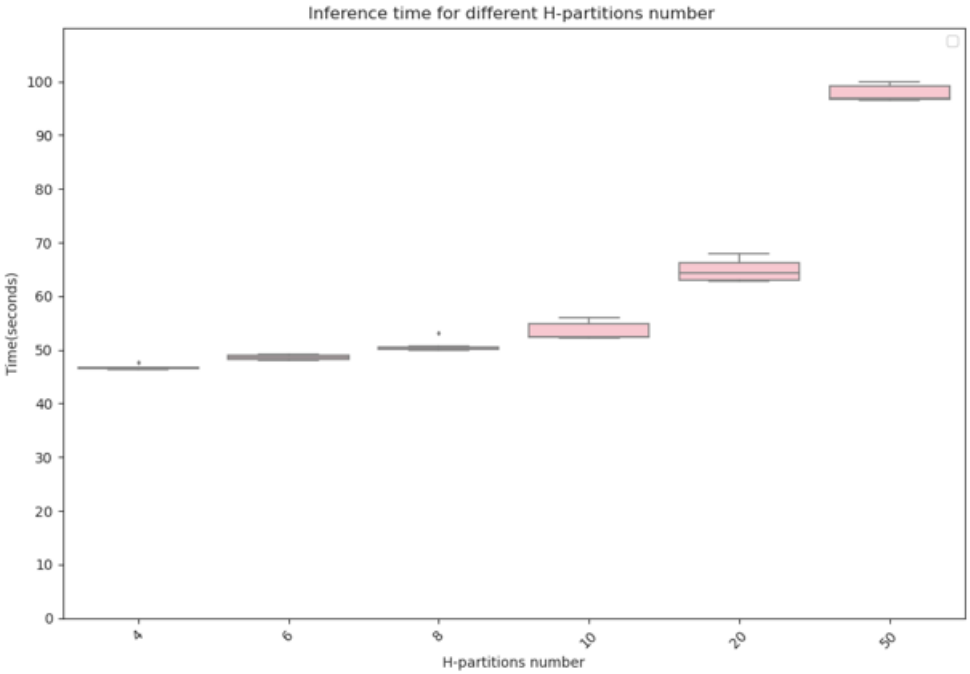

Figure 4.8: Inference time of partitioned VGG16 with different number of H-partitions.

According to the memory constraints, it is mandatory to split the FC1 at least into four H-partitions to be calculated successfully on the available edge device. Each H-partition has a number of parameters that the Raspberry Pi 3 B+ can support.

The next step is to try a different number of H-partitions and observe if it impacts the inference time and the communication overhead. Figure 4.8 shows the results of testing VGG16 inference on a single device with different H-partitions numbers. The box plots show that inference time is relatively constant until 8 H-partitions and then increases exponentially. For 50 H-partitions, the inference time is two times higher than partitioning the FC1 into four H-partitions.

The graph in Figure 4.9 presents the communication overhead for different numbers of H-partitions. The box plots show that communication time is almost the same for the different numbers of H-partitions. For a high number of H-partitions, like 50, the synchronization takes more time than for a low number. Therefore, the inference time variability in Figure 4.8 is related to the synchronization of the H-partitions. The contribution aims to

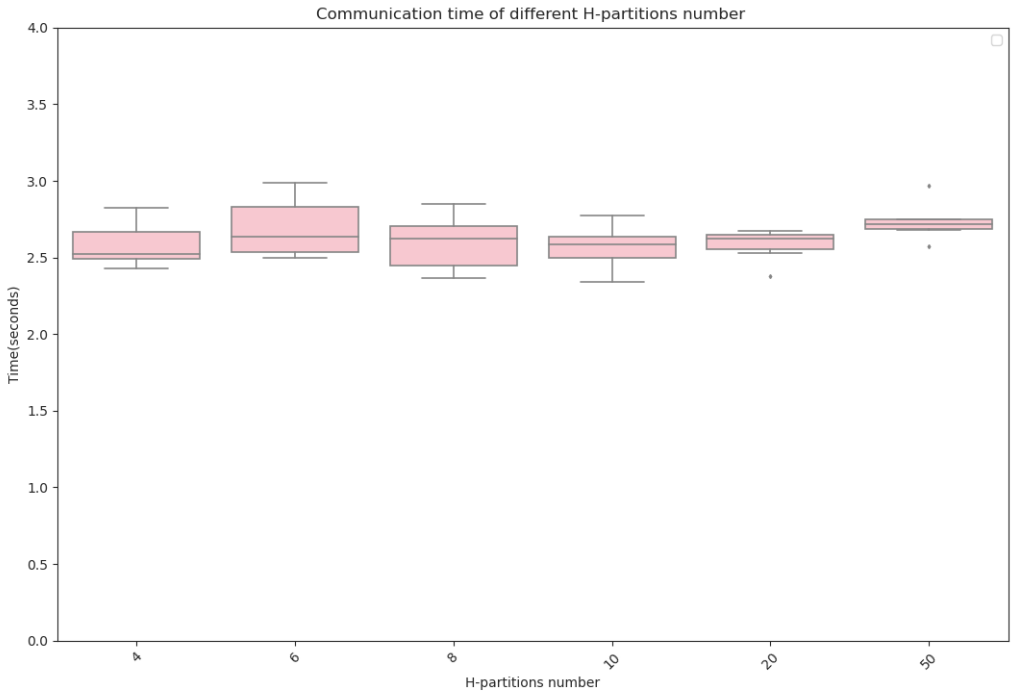

Figure 4.9: Communication overhead of a partitioned VGG16 measured for the FC1.

allow inference with the lowest inference time, so four H-partitions is the optimal number for an inference process on a single-edge device.

These experimental and practical results based on the implemented prototype validate the proposed approach. Indeed, horizontal partitioning leads to high synchronization overhead. All H-partitions need to be fused to obtain the output feature map, which adds synchronization time to the computing time of each H-partitions apart. Therefore, **it is imperative to avoid non-mandatory horizontal partitioning and use horizontal partitioning only when necessary in addition to vertical partitioning**.

HyPS proposes an improved partitioning strategy based on identifying mandatory split points. HyPS allows performing VGG16 inference on an edge device (Raspberry Pi) while avoiding exhaustive use of the device memory, minimizing the communication overhead, and maintaining the same model performance with high image recognition accuracy.

4.3.2.2 Distributed CNN Inference on multiple devices

In this subsection, HyPS allows VGG16 inference on a cluster of edge devices. The partitioned VGG16 inference is distributed across two and three Raspberry Pis. The experimentations aim to compare the inference time and communication overhead between model inference on a single device and across multiple devices. After applying Hybrid partitioning strategy, VGG16 is partitioned into six partitions : two V-partitions denoted Vp1, Vp2 and four H-partitions denoted Hp1, Hp2, Hp3 and Hp4 (see Figure 4.10)

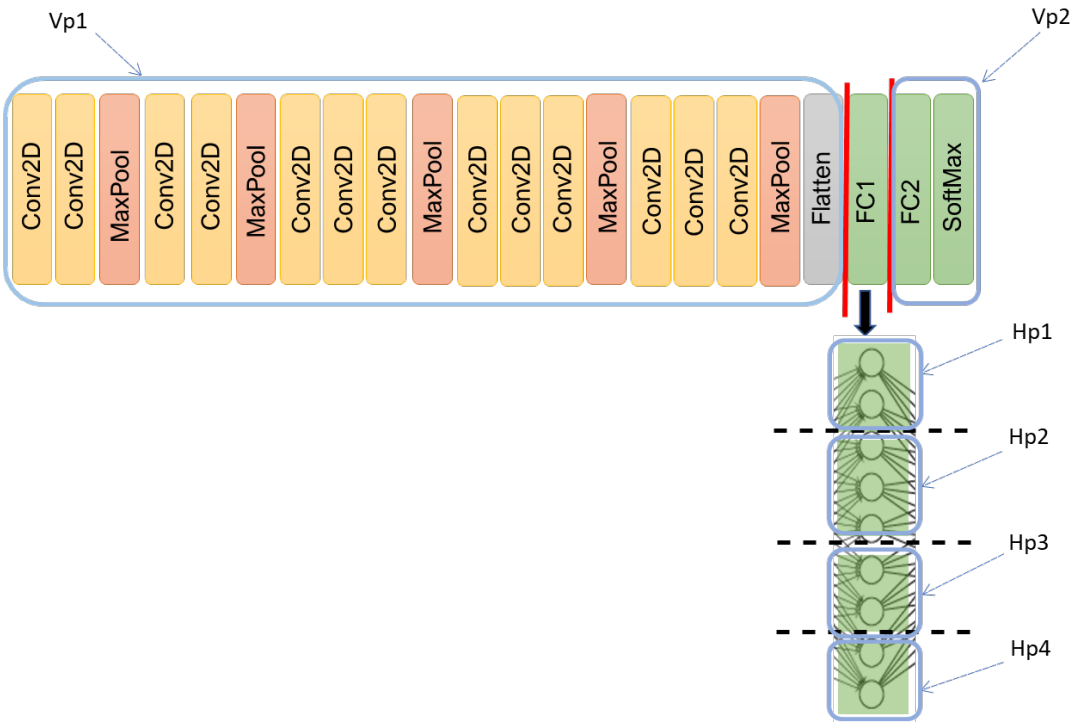

Figure 4.10: VGG16 partitioning using Hybrid partitioning strategy

In this experiment, the manager distributes the V-partitions across one, two or three devices while the H-partitions are executed on a single device. Table 4.3 shows the distribution of the VGG16 partitions for the three execution scenarios.

	Device 1	**Device 2**	**Device 3**
Single device	Vp1, Hp1, Hp2, Hp3, Hp4, Vp2	-	-
Two devices	Vp1	Hp1, Hp2, Hp3, Hp4, Vp2	-
Three devices	Vp1	Hp1, Hp2, Hp3, Hp4	Vp2

Table 4.1: Partition distribution scenarios

Figure 4.11 shows the average inference time and communication overhead of 10 tests of VGG16 inference on single and multiple devices. The tallest bar represents the measurements when running all the model partitions on a single Raspberry Pi. The lowest inference time corresponds to the VGG16 model distributed across three Raspberry Pis. In the HyPS architecture, the manager is responsible of the placement and scheduling of all partitions. The intermediate feature maps exchange is done through the communication hub.

Table 4.2 shows the measurements of VGG16 distributed inference on one, two and three devices. The lowest inference time corresponds to the distributed inference across three Raspberry Pis.

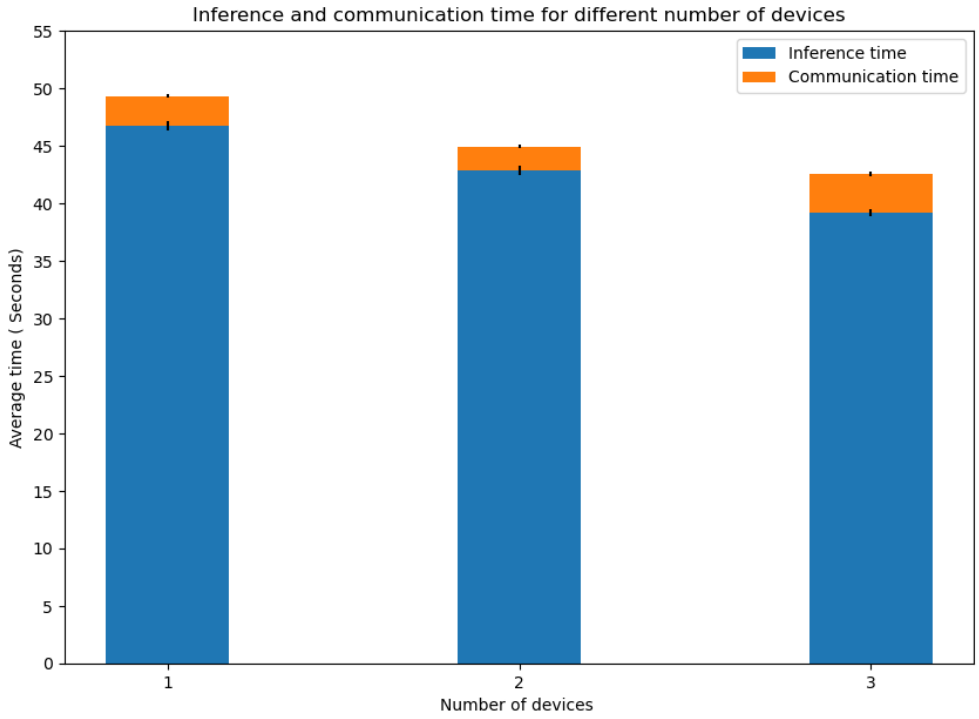

Figure 4.11: Inference time and communication overhead of VGG16 inference distributed across multiple devices.

	Inference time (Seconds)		**Communication time (Seconds)**	
Devices	Average	Standard Deviation	Average	Standard Deviation
1	46.7344	0.4030	2.5832	0.1393
2	42.8802	0.4132	2.1081	0.1476
3	38.7523	0.1476	2.8913	0.1972

Table 4.2: Inference time and communication overhead of VGG16 inference on different devices numbers.

This measurement shows that distributing the V-partitions on multiple devices reduces the inference time and increases a bit the communication time. Therefore, distributing V-partitions while running H-partitions on single machine is better than running all partitions on a single device. This experiment shows that running V-partitions separately reduces the inference time, knowing that the H-partitions are deployed on the same device.

In the following experiment V-partitions are deployed separately in different Raspberry Pis. This experiment aims to compare the H-partitions execution on one, two or three devices. H-partitions represent the first fully connected layer(FC1) partitioned into four parts. Table 4.3 shows an example of H-partitions distribution across one, two and three

	Device 1	**Device 2**	**Device 3**
Single device	Vp1	Hp1, Hp2, Hp3, Hp4	Vp2
Two devices	Vp1, Hp1, Hp2	Hp3, Hp4	Vp2
Three devices	Vp1, Hp1	Hp2, Hp3	Hp4, Vp2

Table 4.3: H-partitions distribution on single and multiple devices

devices. The V-partitions execution is done on different Raspberry Pis.

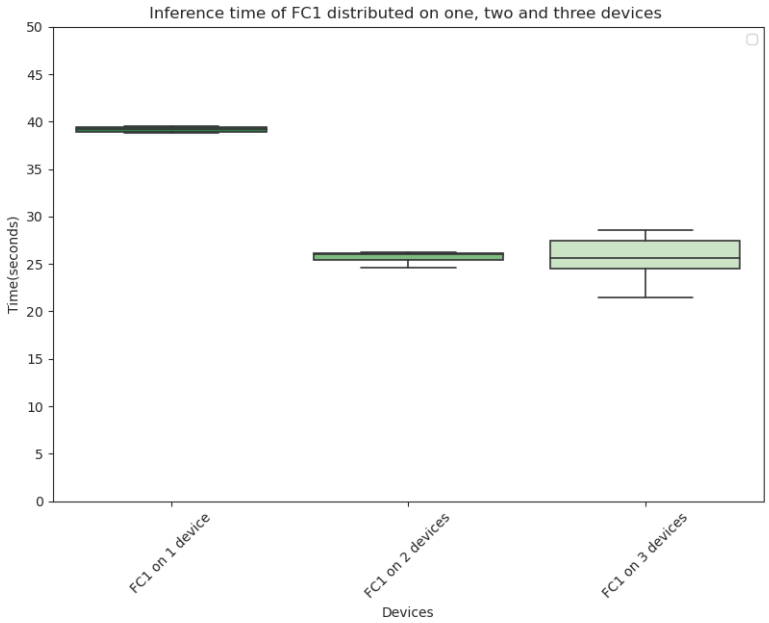

Figure 4.12: Inference time of VGG16 inference with H-partitions distributed across multiple Raspberry Pis.

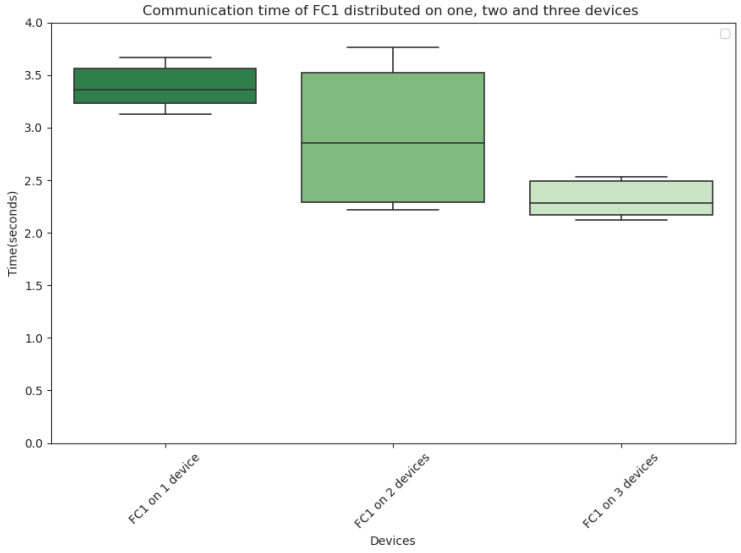

Figure 4.13: Communication overhead of VGG16 inference with H-partitions distributed across multiple Raspberry Pis.

Figure 4.12 shows decreasing inference time when running H-partitions on two and

three devices. Distributing H-partitions on multiple devices permits running H-partitions in parallel, which explains the gain in time and the decrease in inference response time. In Figure 4.13, the box plot in the middle is comparatively tall. This shows that the distribution of measures is different from the other box plots, and communication time is changeable. The median is the average value from a set of values and is shown by the line that divides the box into two parts.

In Figure 4.12, the lowest median corresponds to the FC1 executed in parallel on three Raspberry Pis and also to a thin distribution around this median. The average inference time is lower for 3 Raspberry Pis than for 2 Raspberry Pis itself lower than for 1. However, the distribution on 3 Raspberry Pis leads to a high variation of the inference time than for 2 Raspberry Pis.

According to these experiments, distributing the H-partition execution across multiple devices has a notable impact on inference time unlike the V-partitions distribution. Running H-partitions in separate devices allows parallel execution which allows reducing inference response time and communication overhead. It is important to note that all inference tests provide responses with high accuracy, the model keep the same performance. So, it is recommended to prioritize H-partitions distribution when the target infrastructure is a cluster of multiple devices.

4.4 Conclusion

The evaluation chapter in this manuscript has several benefits. Testing and experimenting with the proposed solution helps to understand better and analyze the actual outcomes. HyPS allows a large CNN inference on a resource-constrained device. The proposed architecture for distributed inference allows reducing inference response time and communication overhead. HyPS maintains the same accuracy as the model before partitioning. HyPS partitions can be deployed easily on edge devices.

Chapter 5

Conclusion

This chapter summarises the thesis work presented in this manuscript. It involves three sections. The first section restates the context of the dissertation, reminds the main findings, briefly resumes the important parts of this manuscript and discusses possible improvements to bring to our solution. The second section highlights the main contributions mentioned in this manuscript and finally, the last section describes some perspectives and interesting research directions that should be carried out in the future.

5.1 Thesis Synopsis

IoT and AI are ubiquitous in several areas. Edge AI paradigm has a significant impact on society in different ways. It surrounds many aspects of life, from connected homes and cities to connected cars and roads. IoT devices generates data that will be gathered to rack user's behavior, make predictions and improve services. The amount of data they make is increasing and may be processed regularly or in real time. Therefore, AI technologies are required to process extensive data and provide decisions while ensuring low latency and data protection.

IoT applications that integrate AI models are computation-intensive. Although current edge devices are increasingly powerful, they are still insufficient to support some complex deep learning models. Performing inference of current AI models using existing solutions raises significant challenges, such as avoiding accuracy loss, avoiding model re-training, and allowing DNN execution on the available infrastructure that could be a single edge device. Existing techniques, such as model compression or any others resulting in model modification, do not address all of these challenges.

To meet these challenges, the thesis work promotes HyPS an efficient, straightforward solution that enables large CNNs inference on resource-constrained device(s). Our solution proposes an hybrid partitioning strategy that allows an efficient CNN splitting by identifying strategic split points on the CNN structure. These strategic split points are used to delimit the generated partitions. HyPS allows CNN inference either on single device or distributed across multiple devices while yielding high-accuracy results. HyPS is a better alternative partitioning strategy that is environmentally friendly, less costly, performant, reliable, and secure.

Throughout this manuscript, we investigate several existing approaches to enable DNNs inference at the edge. State of the art approaches are categorized into two classes. First, model compression techniques that reduce the model size and the memory needed to run a DNN model on edge device. These model compression techniques allow to run DNNs on powerless devices, however, most of them modify the original model structure by removing some parameters or layers, reduce accuracy and require re-training phases to recover accuracy loss.

Second, some works partition the DNN model into small partitions and distribute it across multiple devices. A DNN structure can be partitioned vertically to obtain partitions that include one or more layers. Vertical partitioning, which is layer-wise partitioning, reduces the memory demands per partition.

However, for large DNN models, vertical partitioning strategy is insufficient to get partitions that are small enough to fit resource-constrained devices. Another partitioning strategy refers to split a DNN model at neuron granularity namely horizontal partitioning. This approach allows to generate smaller partitions that fit to edge devices but requires synchronization and communication overhead which increases the inference time. Some IoT applications does not tolerate high latency so horizontal partitioning strategy is not the best solution.

Moreover, most of existing works require distributing the model partitions over a cluster of edge devices to enable inference at the edge and it is not possible to perform complex DNN inference on single device. After a review of state of the art approaches, we introduce our solution HyPS which proposes hybrid partitioning strategy, an architecture and a prototype to orchestrate the inference process of partitioned model. The proposed architecture is used in the implementation and evaluation phase. Hybrid partitioning strategy mixes vertical and horizontal partitioning. Besides, our proposed strategy partitions a large model efficiently and generates necessary and sufficient partitions that can be executed on single and multiple resource-constrained device(s) with high accuracy and without re-training. HyPS allows inference at the edge while reducing communication overhead and latency. The architecture and prototype proposed in HyPS have advantages in terms of reliability, resilience, scalability and privacy.

We implement HyPS on a real testbed to conduct experimentations and assess the performance of our solution. To carry out experiments, we apply HyPS on VGG16 as an example of large CNNs. Experimentation results approve that using HyPS, VGG16 model can be run successfully on a single edge device, which is impossible without hybrid partitioning. Also, VGG16 inference can be distributed across multiple edge devices thanks to the orchestration architecture adopted using HyPS. Experiments allow to validate and reveal the shortcomings of the proposed solution.

Although, HyPS allows to run a large CNN model on resource-constrained device(s), HyPS is facing some challenges which can be improved in the future. For example, HyPS has only been tested on CNNs with chain topology, so we cannot confirm the same efficiency for CNNs with other topologies such as Region-based CNN(R-CNN) [126], ResNet [127] or Multi-Branch Networks like GoogLeNet [128]. Also, our solution requires a prequalification of the IoT infrastructure to get efficient partitioning which depends on the device's number and hardware characteristics. Once the target infrastructure is fixed, it can no longer be modified. HyPS is evaluated on three devices which does not reflect the actual conditions on a real IoT infrastructure where the number of devices can be higher than three. Despite these challenges, HyPS can already handle many use cases and opens up multiple opportunities for future research.

5.2 Contributions

Throughout this thesis, several contributions are made to enable large DNN structures partitioning and deployment at the edge. These contributions are cited as follows:

- a hybrid partitioning strategy that performs partitioning of large CNNs thanks to identifying mandatory and optional split points to successfully split a large CNN model efficiently and run it on resource-constrained device(s) while minimizing inference time and communication overhead,

- an orchestration architecture to enable CNN inference on a single device and allow distributed inference across multiple resource-constrained edge devices. This architecture promotes many advantages such as scalability, reliability, resilience, and data protection,

- a scheduling policy adopted to organize and control data exchanges and partitions execution,

- a proof of concept(PoC) of the hybrid partitioning strategy that allows the generation of model partitions. The PoC allows to validate the proposed partitioning strategy output and ensures the possibility to generate appropriate partitions that can be deployed and executed separately on a single device or distributed across multiple edge devices,

- a prototype allows the implementation of all functionalities and concepts which characterizes HyPS and the proposed architecture.

We consider the criteria grid defined in table 2.4 in Chapter 2 to confirm that our proposed solution meets the limitations of state of the art works. In table 5.1, we used the same evaluation criteria of the table 2.4 :

Proposed approach	**Model structure changes**	**Accuracy loss**	**Model retraining requirement**	**Partitioning type**	**Partitions placement**	**Running on single edge device**
HyPS	No(✔)	No(✔)	No(✔)	Hybrid partitioning	edge	Yes(✔)

Table 5.1: HyPS characteristics

No✔: refers to a positive aspect of the proposed approach.

We evaluate HyPS on a real test bed and provide experimental results analysis. Experiments show that the proposed solution allows inference on resource-constrained devices without any modification of the models' structure. HyPS does not require additional training to recover the accuracy because the accuracy performance is not affected in the inference process. Also, data is processed locally without any transmission to the cloud. These improvements make HyPS a better solution for many real-world use cases and can be integrated into many IoT applications deployments.

5.3 Perspectives and Challenges

Besides the presented contributions provided during this thesis to enable AI models on edge infrastructure, there are other perspectives and potential research directions that can be explored. These perspectives can be categorized into research perspectives and technical perspectives :

- **Geo-distributed placement** The first research perspective consist in applying HyPS on a cluster of heterogeneous and large-scale systems. Hybrid Partitioning strategy and scheduling could be improved by taking into account infrastructure specificities. These specificities can be related to generic capabilities of devices (i.e. RAM, CPU) but also to specific hardware accelerator (e.g. Google Edge TPU), or network characteristics (e.g.: WAN, LAN, wireless connectivity). Also, it is important to study the partitions placement and management on geo-distributed infrastructure [10].

- **Different DNN topologies and types** Applying HyPS to other DNN topologies besides the chain topology is one of the conceivable research perspective. In the presented work, we focus only on chain topology models while in real world use cases there are other CNN models with different topologies such as Multi-Branch Networks like GoogLeNet [128], and ResNet [127]. HyPS can be applied to other types of DNNs, such as Recurrent Neural networks (RNN), Long Short-Term Memory(LSTM), and Generative neural network (GAN). Testing HyPS on different DNN types allow to generalize the use of HyPS on different AI technologies such as speech recognition and natural language processing.

- **Automated parameter setting** HyPS requires as input some parameters such as the CNN structure and the IoT infrastructure characteristics (e.g.threshold). To assign these parameters, a pre-qualification of the input model and the target edge devices is required. As a research perspective, a mechanism that automatically sets these parameters can be developed to improve HyPS.

- **Optimisation of energy consumption** The work presented in this manuscript focuses on reducing communication overhead and inference latency, while several other objectives can be optimized when running DNNs at the edge. One of these objectives is the energy consumption. The integration of IoT devices in smart buildings and cities has many advantages and one of these advantages is to improve energy efficiency and sustainability. Therefore, one of the interesting research objectives is to use AI models to optimize energy consumption and explore green energy to power IoT devices.

- **Batch inference** is an interesting technical perspective because the proposed architecture is dedicated to launch inference on a set of images using the same CNN partitions. In fact, HyPS allows the generation of partitions that can be deployed on edge device. Once the partitions are placed and loaded, our scheduling policy is adapted to retrieve and unstuck the images successively to compute the intermediate feature maps using the same loaded partition. This approach saves the partition transmission and loading times by grouping the inference no longer by image but

by a batch of images for each partition. For the same model partitions and the same orchestration architecture, we can infer extensive data with high accuracy and low latency.

Batch inference using HyPS architecture is highly recommended when predictions must be generated automatically on large-scale datasets. For example, smart farms integrate sensitive physical hardware such as sensors, drones, and bots that monitors and records data, which is then used to get valuable insights. These devices are integrated with DNNs to process data and make insightful decisions. This requires deploying an inference pipeline that can compute several thousand inference jobs on extensive data gathered 24 hours a day. For example, in [129], Cruz et al. IoT system based on computer vision and machine learning technologies to detect and study plant diseases in a smart strawberry farm. The proposed IoT platform involves four parts: Part 1 comprises sensor nodes responsible for collecting the local data, pre-processing, and transferring. Part 2 is composed of collector nodes which are responsible for data processing, organization, storage, and uploading. Cloud services are part 3, and part 4 includes user applications. A camera connected to Raspberry Pi 4 B captures images of strawberry plants in real time, and sensors are used to measure humidity and environment temperature. The object detection CNN chosen for deployment in this system is the Yolo v5s [130] because this version presents a smaller model that can be run on Raspberry Pi 4B. Raspberry Pi 4B boards present limited hardware capacities for training purposes. Therefore, a computer with a powerful Graphic Process Unit (GPU) is required to carry out the training phase. Although the proposed IoT platform allows disease detection and provides accurate results on a strawberry plantation, the model did not detect all diseases and requires improved performance in different lighting profiles and better performance in the IoT infrastructure. An interesting solution to meet these requirements is to integrate HyPS in this platform. First, the Hybrid partitioning strategy allows to run a large model, whatever its size. So, the size constraint is no longer taken into account, and it is possible to perform inference of large CNN on Raspberry Pi 4 B. Second, HyPS architecture allows to carry out batch inference on images of strawberry plants captured by the camera and reduces latency.

- **Implementation on Real IoT application embedding HyPS**. A second technical perspective consist in integrating HyPS into the IoT framework of real use cases. For instance, our solution's evaluation is based on a simple test bed. In the real environment, several external factors may impact the inference process, such as an unexpected breakdown or anonymously collected data. For example, HyPS can be integrated in security cameras and allows image processing in real-time. Performing large CNN model inference on smart camera can improve operations in commercial buildings, protect people and machines from accidents. Thus, real-world deployment of HyPS in a non-controlled situation can definitely be a plus.

In this manuscript, HyPS provides a partitioning strategy that successfully reduces the gap between a computation-intensive DNN and resource-constrained device(s). Our contribution highlights an interesting research direction that aims to explore and enable a wide range of existing trained DNNs to meet IoT infrastructure.

References

[1] Sachin Kumar, Prayag Tiwari, and Mikhail Zymbler. "Internet of Things is a revolutionary approach for future technology enhancement: a review". In: *Journal of Big data* 6.1 (2019), pp. 1–21.

[2] Keyur K Patel and Sunil M Patel. "Professor PSA. Internet of Things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges". In: *Int J Eng Sci Comput* 6.5 (2016), pp. 6122–31.

[3] Mohammad Hasan. "State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally". In: *IoT Analytics* 308 (2022).

[4] Zewen Li et al. "A survey of convolutional neural networks: analysis, applications, and prospects". In: *IEEE transactions on neural networks and learning systems* (2021).

[5] Kaidong Li et al. "Object detection with convolutional neural networks". In: *Deep Learning in Computer Vision*. CRC Press, 2020, pp. 41–62.

[6] H Frank Cervone. "Cloud computing: Pros and cons". In: *Getting Started with Cloud Computing* (2011).

[7] Peshraw Ahmed Abdalla and Asaf Varol. "Advantages to disadvantages of cloud computing for small-sized business". In: *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE. 2019, pp. 1–6.

[8] Ronald L Krutz and Russell Dean Vines. *Cloud security: A comprehensive guide to secure cloud computing*. Wiley Publishing, 2010.

[9] Muhammad Raheel Raza, Asaf Varol, and Nurhayat Varol. "Cloud and fog computing: A survey to the concept and challenges". In: *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE. 2020, pp. 1–6.

[10] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. "An overview of service placement problem in fog and edge computing". In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–35.

[11] Weisong Shi et al. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[12] Wazir Zada Khan et al. "Edge computing: A survey". In: *Future Generation Computer Systems* 97 (2019), pp. 219–235.

[13] Nancy A Angel et al. "Recent advances in evolving computing paradigms: Cloud, edge, and fog technologies". In: *Sensors* 22.1 (2021), p. 196.

[14] Shanhe Yi, Cheng Li, and Qun Li. "A survey of fog computing: concepts, applications and issues". In: *Proceedings of the 2015 workshop on mobile big data*. 2015, pp. 37–42.

[15] Bojana Bajic et al. "EDGE COMPUTING VS. CLOUD COMPUTING: CHALLENGES AND OPPORTUNITIES IN INDUSTRY 4.0." In: *Annals of DAAAM & Proceedings* 30 (2019).

[16] Shuiguang Deng et al. “Edge intelligence: The confluence of edge computing and artificial intelligence”. In: *IEEE Internet of Things Journal* 7.8 (2020), pp. 7457–7469.

[17] Zilong Zhao et al. “Robust anomaly detection on unreliable data”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 630–637.

[18] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and policy considerations for deep learning in NLP”. In: *arXiv preprint arXiv:1906.02243* (2019).

[19] Rikiya Yamashita et al. “Convolutional neural networks: an overview and application in radiology”. In: *Insights into imaging* 9.4 (2018), pp. 611–629.

[20] Jianxin Wu. “Introduction to convolutional neural networks”. In: *National Key Lab for Novel Software Technology. Nanjing University. China* 5.23 (2017), p. 495.

[21] Erqian Tang and Todor Stefanov. “Low-memory and high-performance CNN inference on distributed systems at the edge”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. 2021, pp. 1–8.

[22] Pierre-Emmanuel Novac et al. “Quantization and deployment of deep neural networks on microcontrollers”. In: *Sensors* 21.9 (2021), p. 2984.

[23] Niccoló Nicodemo et al. “Memory Requirement Reduction of Deep Neural Networks Using Low-bit Quantization of Parameters”. In: *arXiv preprint arXiv:1911.00527* (2019).

[24] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. “Degree-quant: Quantization-aware training for graph neural networks”. In: *arXiv preprint arXiv:2008.05000* (2020).

[25] Sumin Kim, Gunju Park, and Youngmin Yi. “Performance Evaluation of INT8 Quantized Inference on Mobile GPUs”. In: *IEEE Access* 9 (2021), pp. 164245–164255.

[26] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[27] Claude Elwood Shannon. “A mathematical theory of communication”. In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[28] Mukhammed Garifulla et al. “A Case Study of Quantizing Convolutional Neural Networks for Fast Disease Diagnosis on Portable Medical Devices”. In: *Sensors* 22.1 (2021), p. 219.

[29] Yunchao Gong et al. “Compressing deep convolutional networks using vector quantization”. In: *arXiv preprint arXiv:1412.6115* (2014).

[30] Aojun Zhou et al. “Incremental network quantization: Towards lossless cnns with low-precision weights”. In: *arXiv preprint arXiv:1702.03044* (2017).

[31] Ron Banner, Yury Nahshan, and Daniel Soudry. “Post training 4-bit quantization of convolutional networks for rapid-deployment”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[32] Yoni Choukroun et al. "Low-bit quantization of neural networks for efficient inference". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3009–3018.

[33] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[34] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[35] Haoli Bai et al. "Towards efficient post-training quantization of pre-trained language models". In: *arXiv preprint arXiv:2109.15082* (2021).

[36] Rui Wang et al. "A Real-Time Object Detector for Autonomous Vehicles Based on YOLOv4". In: *Computational Intelligence and Neuroscience* 2021 (2021).

[37] M Mary Shanthi Rani et al. "DeepCompNet: A Novel Neural Net Model Compression Architecture". In: *Computational Intelligence and Neuroscience* 2022 (2022).

[38] Karthik Abinav Sankararaman et al. "The impact of neural network overparameterization on gradient confusion and stochastic gradient descent". In: *International conference on machine learning*. PMLR. 2020, pp. 8469–8479.

[39] Arash Ardakani, Carlo Condo, and Warren J Gross. "Sparsely-connected neural networks: towards efficient vlsi implementation of deep neural networks". In: *arXiv preprint arXiv:1611.01427* (2016).

[40] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured pruning of deep convolutional neural networks". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–18.

[41] Shaohui Lin et al. "Towards optimal structured cnn pruning via generative adversarial learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2790–2799.

[42] Song Han et al. "Learning both weights and connections for efficient neural network". In: *Advances in neural information processing systems* 28 (2015).

[43] Anthony Berthelier et al. "Deep model compression and architecture optimization for embedded systems: A survey". In: *Journal of Signal Processing Systems* 93.8 (2021), pp. 863–878.

[44] Tianyun Zhang et al. "A systematic dnn weight pruning framework using alternating direction method of multipliers". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 184–199.

[45] Hao Li et al. "Pruning filters for efficient convnets". In: *arXiv preprint arXiv:1608.08710* (2016).

[46] Yu Cheng et al. “A survey of model compression and acceleration for deep neural networks”. In: *arXiv preprint arXiv:1710.09282* (2017).

[47] C Bucilua, R Caruana, and A Niculescu-Mizil. “Model compression, in proceedings of the 12 th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining”. In: *New York, NY, USA* 3 (2006).

[48] Guo-Hua Wang, Yifan Ge, and Jianxin Wu. “Distilling knowledge by mimicking features”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[49] Sergey Zagoruyko and Nikos Komodakis. “Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer”. In: *arXiv preprint arXiv:1612.03928* (2016).

[50] Adriana Romero et al. “Fitnets: Hints for thin deep nets”. In: *arXiv preprint arXiv:1412.6550* (2014).

[51] Guobin Chen et al. “Learning efficient object detection models with knowledge distillation”. In: *Advances in neural information processing systems* 30 (2017).

[52] Paul Bergmann et al. “Uninformed students: Student-teacher anomaly detection with discriminative latent embeddings”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 4183–4192.

[53] Victor Sanh et al. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter”. In: *arXiv preprint arXiv:1910.01108* (2019).

[54] Ying Zhang et al. “Deep mutual learning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4320–4328.

[55] Li Yuan et al. “Revisiting knowledge distillation via label smoothing regularization”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 3903–3911.

[56] Jimmy Ba and Rich Caruana. “Do deep nets really need to be deep?” In: *Advances in neural information processing systems* 27 (2014).

[57] Lin Wang and Kuk-Jin Yoon. “Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[58] Tara N Sainath et al. “Low-rank matrix factorization for deep neural network training with high-dimensional output targets”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6655–6659.

[59] Misha Denil et al. “Predicting parameters in deep learning”. In: *Advances in neural information processing systems* 26 (2013).

[60] Emily L Denton et al. “Exploiting linear structure within convolutional networks for efficient evaluation”. In: *Advances in neural information processing systems* 27 (2014).

[61] Cheng Tai et al. "Convolutional neural networks with low-rank regularization". In: *arXiv preprint arXiv:1511.06067* (2015).

[62] Xiangyu Zhang et al. "Efficient and accurate approximations of nonlinear convolutional networks". In: *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*. 2015, pp. 1984–1992.

[63] Matan Ben Noach and Yoav Goldberg. "Compressing pre-trained language models by matrix decomposition". In: *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*. 2020, pp. 884–889.

[64] Mengnan Du et al. "What do compressed large language models forget? robustness challenges in model compression". In: *arXiv preprint arXiv:2110.08419* (2021).

[65] Canwen Xu and Julian McAuley. "A survey on model compression for natural language processing". In: *arXiv preprint arXiv:2202.07105* (2022).

[66] Honglei Zhang, Serkan Kiranyaz, and Moncef Gabbouj. "Finding better topologies for deep convolutional neural networks by evolution". In: *arXiv preprint arXiv:1809.03242* (2018).

[67] Xianzhong Tian et al. "Mobility-included DNN partition offloading from mobile devices to edge clouds". In: *Sensors* 21.1 (2021), p. 229.

[68] Nicos Christofides and P Brooker. "The optimal partitioning of graphs". In: *SIAM Journal on Applied Mathematics* 30.1 (1976), pp. 55–69.

[69] Eric SH Wong, Evangeline FY Young, and Wai-Kei Mak. "Clustering based acyclic multi-way partitioning". In: *Proceedings of the 13th ACM Great Lakes symposium on VLSI*. 2003, pp. 203–206.

[70] Shuai Zhang et al. "Deepslicing: collaborative and adaptive cnn inference with low latency". In: *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), pp. 2175–2187.

[71] Hyuk-Jin Jeong et al. "IONN: Incremental offloading of neural network computations from mobile devices to edge servers". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 401–411.

[72] Chuang Hu et al. "Dynamic adaptive DNN surgery for inference acceleration on the edge". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1423–1431.

[73] Beibei Zhang et al. "Dynamic DNN Decomposition for Lossless Synergistic Inference". In: *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2021, pp. 13–20.

[74] Huaming Wu et al. "An optimal offloading partitioning algorithm in mobile cloud computing". In: *International Conference on Quantitative Evaluation of Systems*. Springer. 2016, pp. 311–328.

[75] Chenghao Hu and Baochun Li. "Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices". In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 330–339.

[76] Chongwu Dong et al. "Joint Optimization With DNN Partitioning and Resource Allocation in Mobile Edge Computing". In: *IEEE Transactions on Network and Service Management* 18.4 (2021), pp. 3973–3986.

[77] Tao Zheng et al. "A survey of computation offloading in edge computing". In: *2020 International Conference on Computer, Information and Telecommunication Systems (CITS)*. IEEE. 2020, pp. 1–6.

[78] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. "Edge machine learning for ai-enabled iot devices: A review". In: *Sensors* 20.9 (2020), p. 2533.

[79] Congfeng Jiang et al. "Toward computation offloading in edge computing: A survey". In: *IEEE Access* 7 (2019), pp. 131543–131558.

[80] Yiping Kang et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge". In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 615–629.

[81] Rishabh Mehta and Rajeev Shorey. "Deepsplit: Dynamic splitting of collaborative edge-cloud convolutional neural networks". In: *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE. 2020, pp. 720–725.

[82] Raby Hamadi et al. "A Hybrid Artificial Neural Network for Task Offloading in Mobile Edge Computing". In: *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE. 2022, pp. 1–4.

[83] En Li et al. "Edge AI: On-demand accelerating deep neural network inference via edge computing". In: *IEEE Transactions on Wireless Communications* 19.1 (2019), pp. 447–457.

[84] Changqing Luo et al. "Energy-efficient autonomic offloading in mobile edge computing". In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/Cyb* IEEE. 2017, pp. 581–588.

[85] Bo Yang et al. "Offloading optimization in edge computing for deep-learning-enabled target tracking by internet of UAVs". In: *IEEE Internet of Things Journal* 8.12 (2020), pp. 9878–9893.

[86] Haneul Ko and Sangheon Pack. "Distributed device-to-device offloading system: Design and performance optimization". In: *IEEE Transactions on Mobile Computing* 20.10 (2020), pp. 2949–2960.

[87] Xu Chen et al. "Exploiting massive D2D collaboration for energy-efficient mobile edge computing". In: *IEEE Wireless communications* 24.4 (2017), pp. 64–71.

[88] Teemu Leppänen and Jukka Riekki. "Energy efficient opportunistic edge computing for the Internet of Things". In: *Web Intelligence*. Vol. 17. 3. IOS Press. 2019, pp. 209–227.

[89] Mahadev Satyanarayanan et al. "The case for vm-based cloudlets in mobile computing". In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[90] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2348–2359.

[91] Jiachen Mao et al. "Modnn: Local distributed mobile computing system for deep neural network". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1396–1401.

[92] Rafael Stahl et al. "DeeperThings: Fully distributed CNN inference on resource-constrained edge devices". In: *International Journal of Parallel Programming* 49.4 (2021), pp. 600–624.

[93] Cian-You Yang et al. "Cooperative distributed deep neural network deployment with edge computing". In: *ICC 2021-IEEE International Conference on Communications*. IEEE. 2021, pp. 1–6.

[94] Li Zhou et al. "Distributing deep neural networks with containerized partitions at the edge". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[95] Xueyu Hou et al. "Distredge: Speeding up convolutional neural network inference on distributed edge devices". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2022, pp. 1097–1107.

[96] Zhipeng Gao et al. "EdgeSP: Scalable Multi-Device Parallel DNN Inference on Heterogeneous Edge Clusters". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2021, pp. 317–333.

[97] Siqi Wang et al. "High-Throughput CNN Inference on Embedded ARM big.LITTLE Multi-Core Processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP (Sept. 2019), pp. 1–1. DOI: 10.1109/TCAD.2019.2944584.

[98] Hung-Yang Chang et al. "PipeBERT: High-throughput BERT Inference for ARM Big. LITTLE Multi-core Processors". In: *Journal of Signal Processing Systems* (2022), pp. 1–18.

[99] Arjun Parthasarathy and Bhaskar Krishnamachari. "DEFER: Distributed Edge Inference for Deep Neural Networks". In: *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE. 2022, pp. 749–753.

[100] Thaha Mohammed et al. “Distributed inference acceleration with adaptive DNN partitioning and offloading”. In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 854–863.

[101] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. “Distributed deep neural networks over the cloud, the edge and end devices”. In: *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*. IEEE. 2017, pp. 328–339.

[102] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. “JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services”. In: *IEEE Transactions on Mobile Computing* 20.2 (2019), pp. 565–576.

[103] Zhiming Hu et al. “Deephome: Distributed inference with heterogeneous devices in the edge”. In: *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*. 2019, pp. 13–18.

[104] Abdullah Lakhan et al. “Deep neural network-based application partitioning and scheduling for hospitals and medical enterprises using IoT assisted mobile fog cloud”. In: *Enterprise Information Systems* 16.7 (2022), p. 1883122.

[105] Qunsong Zeng et al. “Energy-efficient radio resource allocation for federated edge learning”. In: *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE. 2020, pp. 1–6.

[106] Fabíola Martins Campos de Oliveira and Edson Borin. “Partitioning convolutional neural networks to maximize the inference rate on constrained IoT devices”. In: *Future Internet* 11.10 (2019), p. 209.

[107] Jung Hwan Kim, Alwin Poulose, and Dong Seog Han. “The Customized Visual Geometry Group Deep Learning Architecture for Facial Emotion Recognition”. In: *Available at SSRN 4087604* ().

[108] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[109] Max Ferguson et al. “Automatic localization of casting defects with convolutional neural networks”. In: *2017 IEEE international conference on big data (big data)*. IEEE. 2017, pp. 1726–1735.

[110] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[111] Rohit Thakur. “Step by step VGG16 implementation in Keras for beginners”. In: *Medium* (2019).

[112] Mário P Véstias. “A survey of convolutional neural networks on edge with reconfigurable computing”. In: *Algorithms* 12.8 (2019), p. 154.

[113] Umar Ozeer et al. “F3ARIoT: A framework for autonomic resilience of IoT applications in the Fog”. In: *Internet of Things* 12 (2020), p. 100275.

[114] Loic Letondeur, François-Gaël Ottogalli, and Thierry Coupaye. "A demo of application lifecycle management for IoT collaborative neighborhood in the Fog: Practical experiments and lessons learned around docker". In: *2017 IEEE Fog World Congress (FWC)*. IEEE. 2017, pp. 1–6.

[115] Yifan Wang et al. "{HydraOne}: An Indoor Experimental Research and Education Platform for {CAVs}". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. 2019.

[116] Tianze Wu et al. "HydraMini: An FPGA-based affordable research and education platform for autonomous driving". In: *2020 International Conference on Connected and Autonomous Driving (MetroCAD)*. IEEE. 2020, pp. 45–52.

[117] Victor Wikén. *An Investigation of Low-Rank Decomposition for Increasing Inference Speed in Deep Neural Networks With Limited Training Data*. 2018.

[118] Roger A Light. "Mosquitto: server and client implementation of the MQTT protocol". In: *Journal of Open Source Software* 2.13 (2017), p. 265.

[119] Ivan Vaccari, Maurizio Aiello, and Enrico Cambiaso. "SlowITe, a novel denial of service attack affecting MQTT". In: *Sensors* 20.10 (2020), p. 2932.

[120] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[121] François Chollet et al. *keras*. 2015.

[122] MV Masdani and Denny Darlis. "A comprehensive study on MQTT as a low power protocol for internet of things application". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 434. 1. IOP Publishing. 2018, p. 012274.

[123] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. "MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks". In: *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE. 2008, pp. 791–798.

[124] Patrick Th Eugster et al. "The many faces of publish/subscribe". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[125] Dazhi Chen and Pramod K Varshney. "QoS support in wireless sensor networks: a survey." In: *International conference on wireless networks*. Vol. 233. 2004, pp. 1–7.

[126] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[127] Muhammad Shafiq and Zhaoquan Gu. "Deep residual learning for image recognition: A survey". In: *Applied Sciences* 12.18 (2022), p. 8972.

[128] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[129] Mateus Cruz et al. “Smart Strawberry Farming Using Edge Computing and IoT”. In: *Sensors* 22.15 (2022), p. 5866.

[130] Zijian Wang et al. “Fast personal protective equipment detection for real construction sites using deep learning approaches”. In: *Sensors* 21.10 (2021), p. 3478.