Hybrid Partitioning for Embedded and Distributed CNNs Inference on Edge Devices

Nihel Kaboubi, Loïc Letondeur, Thierry Coupaye, Frédéric Desprez, Denis Trystram

Hybrid Partitioning for Embedded and Distributed CNNs Inference on Edge Devices

Nihel Kaboubi[1,2][0000−0002−1538−0785], Loïc Letondeur[1][0000−0003−0817−3689], Thierry Coupaye[1], Fréderic Desprez[2], and Denis Trystram[2]

[1] Orange, France
nihel.kaboubi@orange.com
loic.letondeur@orange.com
thierry.coupaye@orange.com
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
frederic.desprez@inria.fr
denis.trystram@inria.fr

Abstract. Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs) are ubiquitously utilized in many Internet of Things applications, especially for real-time image-based analysis. In order to cope with concerns such as resiliency, privacy and near real time analysis, these models must be deployed on edge devices. Particularly for large models, the large number of parameters becomes a bottleneck for the inference process because edge devices are resource constrained, subjects to failures and/or hardware faults. New solutions to cope with these issues are required. This paper proposes a hybrid partitioning strategy, architecture and implementation (called HyPS), which identifies the best positions in the model structure to split the network structure into small partitions that fit resources constraints of edge devices noticeably by decreasing instantaneous memory needs. The generated partitions consume less memory than the original network and each partition can be processed almost separately, resulting in new ways to process CNN's execution at the edge. Thanks to this partitioning strategy, large CNNs inference can be run without modifying the main model architecture. The proposed approach is assessed on the well-known neural network structure of VGG16 for image classification. The results of the experimental campaign show that the partitioning method allows for the successful inference of large models on devices with limited overhead and high accuracy.

Keywords: Distributed Inference · Edge Computing · Edge Intelligence · Convolutional Neural Networks · Internet of Things

1 Introduction

Internet of Things (IoT) and Artificial Intelligence (AI) affect our daily lives in a revolutionary way. Billions of connected devices should be deployed in homes, buildings, vehicles, cities, and industries. Connected devices product data and

offer interactions used to enhance smart services in their surrounding environment. AI models can be used and are based on DNNs and noticeably on CNNs on various tasks such as image recognition, video analysis, or object detection.

CNNs are typically deployed on remote cloud servers, requiring the upload of data through all the communication infrastructure. This can be an issue regarding infrastructure solicitation and latency. Such uploads are also problematic concerning privacy [1] [2] as data are shared with third-party and hackers might access this information along the way [3]. Performing inference in the cloud can be a problem for some critical applications for a big telecommunications operator, which nevertheless has an extensive infrastructure at the network periphery (e.g. : relay antennas, network point of presence (POPs), internet access boxes, etc.) [4].

A better alternative solution is to relocate CNN models at the edge of the network. This process is known as Edge intelligence, i.e., Edge computing applied to AI [5]. The edge intelligence paradigm moves computing resources from clouds and data centers as close as possible to the originating data source. Edge infrastructure is composed of one or more edge devices that will leverage deep learning models to implement accurate predictions, make decisions, and decode behavior behind sensors' data.

However, inferring pre-trained large CNNs consumes significant time, memory, and computational resources that can be higher than most edge devices' capabilities. Some existing works use model compression techniques [6][7] to adapt CNN models to run at the edge. Others use partitioning strategies to split CNN's structures into small partitions that fit edge devices [8][9]. Besides hardware capabilities, edge devices often suffer from failures that result in unpredictable service loss because edge devices are often located in unprotected areas (e.g., customers' homes). Another tremendous topic related to edge intelligence is the protection of secrets. This issue concerns data to process and the used AI models themselves. If various approaches intent to fill the gap between resources demands of CNNs models and edge devices capabilities for inference, they do not cover the subject as proposed in this paper, that is without impacting accuracy, without requiring to re-train a model and/or without complex computations before deployment. The proposed solution also brings preliminary answers concerning secrets protection and execution reliability. Other existing approaches only partially cover the described problems.

The main contributions of this work are listed below:

- **a strategy which couples different partitioning methods. It offers the opportunity to effectively partition a large CNN model by identifying the best partitioning points while minimizing the communication cost and inference response latency**. This strategy allows inference processing using large CNNs without modifying their main architecture and guarantees high accuracy. **Partitioned CNNs can be easily executed on one device or can be distributed across a cluster of multiple edge devices**,

- **an orchestration architecture** for hybrid distributed inference. This architecture also exhibits good properties in terms of reliability, resilience, and privacy,
- **a prototype** that demonstrates the functional behavior of the proposed concepts of HyPS assessed by **experimental results on an actual testbed**.

The paper is organized as follows. Section 2 presents the background, context and objectives of this work, and introduces illustrative use cases which will be used throughout the article. Section 3 presents the proposed solution and architecture overview. Section 4 discusses the evaluation of HyPS on a defined testbed and the analysis of experimental results. Section 5 briefly reviews the relevant existing researches related to the proposed work. Section 6 concludes and introduces some perspectives for future works.

2 Background, Context, and Objectives

2.1 Overview of CNNs

A CNN specializes in processing data with a grid-like topology, such as an image. CNN architecture is essentially composed of two parts: *feature extractor* and *classifier*. Feature extractor layers process the original input, and classifier layers then classify the resultant features. A CNN model mainly includes *Convolution layers (Conv), Pooling layers (Pool), Batch Normalization layers (BN)*, and *Fully Connected layers (FC)*. A CNN can be deployed monolithically on a single-edge device or partitioned into multiple partitions. CNNs have a chain topology as each layer only depends on the previous layer's output and not on the other layers. This characteristic will help us to distribute partitions on one or multiple devices and sequentially perform inference, which will be detailed later.

A well-known CNN example is VGG16 [10], used as an illustrative example in the rest of this article. Visual Geometry Group (VGG) is a popular and clear-in-structure CNN model that includes all mainstream layer types. VGG16 (cf. Figure 1) is trained with 16 layers consisting of 13 convolution layers and three fully connected layers.

VGG architecture has been retained in this article because it has proven to be a reference in many complex tasks such as object detection, image recognition and image classification. Noticeably, VGG16 achieves 92.7% top-5 test accuracy in ImageNet [11], a dataset of over 14 million images belonging to 1000 classes. We chose VGG16 as a representative architecture of CNN models. VGG16 represents a great potential for applications in several real-world use cases such as smart factory [12] and autonomous vehicles [13]. Therefore, performing VGG16 inference at the edge can benefit to many IoT applications.

2.2 Context

The VGG16 model needs more than 520MB in memory and requires 15.5G multiply-accumulate operations [14] to classify one image with a 224 *224 resolution. The training of a VGG model can require 2-3 weeks [15] on advanced

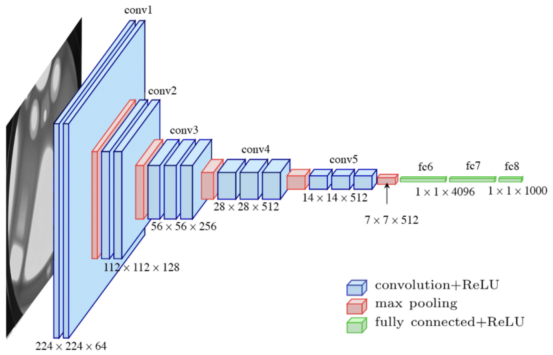

Fig. 1: VGG16 architecture.(Source: Researchgate.net)

GPUs. Because subsequent CO_2 emissions have an impact on the environment [16], the proposed approach aims at capitalizing on the AI models existing training without requiring any re-training phase. Even during inference, these pre-trained models often present a high computational cost, that prevents their execution on constrained devices. In this work, we will use a specific strategy that allows performing inference of partitioned VGG16 on edge infrastructure while minimizing induced overhead and solving resource constraints issue.

2.3 Objective: CNN Model Partitioning Strategies

CNN inference is usually computation-intensive, especially when CNN models are large. They require a lot of memory to run inference calculations but many edge devices do not have enough resources. To this, many previous works have proposed strategies for deploying CNNs models on resource-constrained devices. Besides following concepts and definitions, further details are given in the related works section.

Among existing kinds of strategies,*partitioning* refers to the splitting of an entire model on specific locations to obtain one or more partitions. A first partitioning strategy is *data partitioning*: dividing the input data given to every CNN layer into several small partitions. Weight partitioning is not considered, and each partition includes all layers of the CNN model. The input data to layer L_i of partition P_i may have to be shared with layers from other partitions. After the computation of all partitions, outputs may be fused to get the final inference output. This does not decrease memory requirements and implies a high complexity to synchronize all the inference.

A second partitioning strategy is *vertical model partitioning* which consists in building partitions made of one or several complete layers named V-partition. In Fig 2, partitions are represented by rectangles colored in blue, and each of them includes one or a group of consecutive layers. Each layer keeps weights and parameters fixed during the training phase. Going deeper into the network,

the complexity increases, and filters are applied to identify more prominent elements. Vertical partitioning strategy is easier to deploy and run as pre-trained models can be divided into multiple layer groups and distributed to different edge nodes. Nevertheless, this strategy could not be sufficient to decrease the memory footprints of partitions as they are a combination of entire layers.

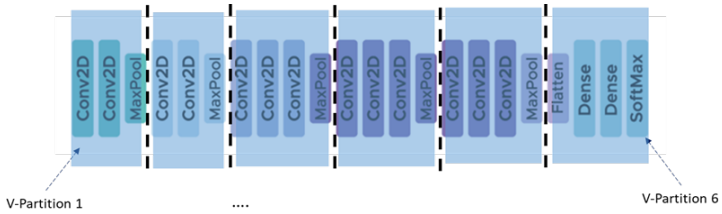

Fig. 2: Vertical partitioning.

A third partitioning strategy is *horizontal model partitioning* which partitions the weights across layers. A partition can include one or more units from different layers. The input data are sent to all partitions. All partitions must communicate and synchronize with each other because all the output data must be concatenated with the output data from the different partitions to get the final output. The horizontal strategy is efficient regarding memory footprint limitation for each partition, but at the cost of a high complexity concerning deployment and execution, as it requires more synchronizations than the vertical strategy.

This article proposes to combine advantages of existing partitioning strategies and more precisely to combine vertical and horizontal partitioning. Because only these two strategies can reduce memory requirements, a heuristic is also proposed: it mainly consists in using vertical partitioning as a bootstrap for distributed execution of CNNs and then horizontal partitioning but only when needed and on a single partition at a time. As shown further in this article, this approach permits to achieve good performances, enable reliable execution of a CNN on a typical edge infrastructure and open new use cases.

2.4 Illustrative Real Life Use Cases

The hybrid partitioning strategy proposed in this article is helpful in several real-life use cases. For example, consider an AI service company sells an image recognition system linked to surveillance cameras. This system can be integrated into any embedded device that composes a typical Telecommunication company.

For instance, an image recognition task can be performed on an internet gateway. Such a device can not run many CNNs by itself without optimizations. Our proposed strategy addresses resource constraints by distributing inference in time on a single device or multiple devices. Model partitioning reduces the size

of the tasks assigned at a time to each device, reducing memory consumption and avoiding calculation problems.

Because internet gateways can also be abruptly unplugged, all the processed data can be lost, and the inference must then be restarted from scratch. In our solution, intermediate inference outputs after each partition are saved, avoiding restarting calculations from scratch. Interrupted calculations can thus be resumed making inference more reliable on edge devices.

A second use case is that of a smart factory, a digitized manufacturing facility that uses connected devices, machinery, and production systems to collect and share data continuously. Image recognition models are used to improve the productivity and detect defective pieces rapidly. This smart factory is composed of different production areas. Each area takes charge of a part in the manufacturing process. The available machines in the production are dedicated to do a specific production task and are very constrained in memory and computing capacity.

Generally, the manufacturer wants to avoid buying more powerful machines to classify images but instead takes advantage of their existing machines. These machines must give a real-time response. So, running the whole AI model on one machine can increase the computational load, cause memory saturation, and increase subsequent response latency. HyPS solves this problem by sequentially running a partitioned model on one or multiple machines. Indeed, the execution of one partition will be less expensive than the whole model in one block. The collected data are used locally and are not accessible by other production units. So, the proposed strategy helps the factory to improve the quality of construction pieces without slowing the production process nor causing damages to the machinery. It also permits to preserve data and model privacy since each machine only deals with specific partitions of the global model. The following sections present the proposed partitioning strategy and discuss the architecture details for HyPS implementation.

3 Proposed Approach: Vertical, Horizontal, and Hybrid Partitioning Strategies

The methods of partitioning a network have been already proposed in previous papers, but the novelty here is the strategy of mixing two ways of partitioning (vertical and horizontal). This technique aims at deploying trained CNN model on edge device(s) without any modification in the entire architecture or loss in accuracy while optimizing the computation time as much as possible.

The main objective of this contribution is to capitalize on the AI models previous training phase, use it actively on inference and avoid the re-training phase as much as possible. In section 3.1, the vertical partitioning method is introduced and presented over different cases. In section 3.2 , a second existing way of partitioning, namely horizontal partitioning, is presented. Finally, Section 3.3 contains the new partitioning strategy and the benefits behind coupling two strategies of partitioning.

3.1 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 generates its feature map. Dimensions of feature maps produced by the output layer can vary considerably, resulting in a possible huge communication. 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 or the feature map output of the prior layers, and the type of the layer.

As mentioned in subsection 2.1, all layers in CNNs are arranged following a specific pattern. The output feature map of a convolutional layer is a complex matrix with high dimensions. After each convolution block, there is a pooling layer. This layer performs the dimensionality reduction on the input by reducing the number of parameters. 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 ensure feature map transmission between partitions. This process generates high communication costs if the transmitted data are extensive and may increase latency. Fortunately, the communication charge in a vertical partitioning strategy is less than the communication cost in a horizontal partitioning strategy, as explained in the following subsection.

3.2 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 the neurons granularity. Horizontal partitioning is the thinnest way of partitioning. Inside a model, layers have different number of parameters and for some layers this number of parameters can be very high. Those layers demand a powerful computing capacity to generate feature map. For limited resources devices, it could be impossible to perform calculations of complex layers in one operation. So, vertical partitioning alone is insufficient to perform CNN inference efficiently on edge devices.

For example, our experiments showed it was impossible to run the VGG16 model without horizontal partitioning on Raspberry Pi 3B+ with 1 GB of RAM. This specific layer has a number of parameters that exceeds one hundred two million ($102 * 10^6$) parameters which is too high for a device as a Raspberry Pi. The horizontal partitioning strategy splits a given complex layer into small groups of neurons, whereas the input data layer is not partitioned.

In Figure 3, the complex layer in the VGG16 structure is partitioned into four H-partitions which are represented by a green rectangle (four H-partitions are chosen to simplify the presentation in the figure). In this case, partitioning one layer into H-partitions reduces the number of parameters, storage needs,

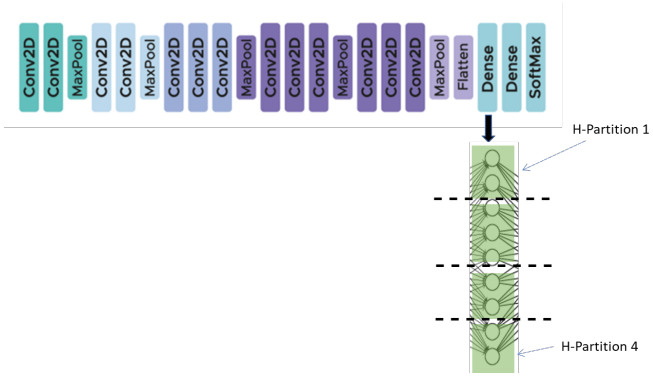

Fig. 3: Horizontal partitioning of the first fully connected layer in the VGG16 structure.

and the memory required to compute layer features. Therefore, the CNN layer that includes intensive computations is divided into many H-partitions. Each H-partition is only responsible for computing a part of the output of the current layer. At the same time, a specific algorithm collects and merges all the output's H-partitions to get the full feature map before executing the next layer.

Partial features maps collection and merging is a synchronization process that introduces a synchronization cost. This cost does not exist for vertical partitioning, so to preserve performance, horizontal partitioning must be used as few as possible. The following section describes how HyPS takes the highest benefits from both sorts of partitioning.

3.3 Hybrid Partitioning Strategy

Customers and industrial users need help choosing the best partitioning strategy that enables large CNN model inference on resource-constrained device(s). A preliminary decision support system that provides a guided partitioning strategy strategy is proposed. HyPS takes input information about the global model structure to be partitioned and the characteristics of the target edge infrastructure. Afterward, 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 partition. For a large CNN model like VGG16, vertical partitioning generates V-partitions that are still computationally intensive and complex to be executed on a edge device(s). On the other hand, 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 H-partitions must be restricted to avoid ineffective operations. The intended solution aims to minimize computational costs

related to H-partitions synchronization and prevents 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, if it is mandatory, to perform the CNN partition execution on the target edge infrastructure. The proposed algorithm starts by going through all the model layers one by one and checks if it is possible to run it on the edge device. The program runs this process until reaching a complex layer which results in three vertical partitions for one particular complex layer: one V-partition before the first added mandatory split point, a second after the second mandatory split point and a third constituted of the complex layer itself and alone. Then, the third V-partition containing the complex layer is itself partitioned into H-partitions as small as required to fit targeted execution infrastructure. HyPS operates H-partitioning when required and on a single layer at a time, to preserve original model performances. Algorithm 1 allows to identify the mandatory partition points for an input CNN model.

Algorithm 1 Get mandatory split points

Require:
Model: CNN model
Threshold : maximum number of parameters supported by the device
Ensure: *LM* : list of total mandatory split points
for each layer in *model* **do**
  if number of parameters > Threshold **then**
    LM ← Mandatory Split Points
  else if the last layer **then**
    return *LM*
  end if
end for

After a first step consisting in the identification of the possible mandatory split points, HyPS identifies optional split points in a second step. Optional split points are particular locations in the NN architecture where partitioning operation allows to obtain smaller V-partitions to cope with particular needs related to the use of the NN (e.g. privacy concerns). HyPS identifies all pooling layers as so called *Optional Split Points*. The benefits of using pooling layers include reducing the complexity, speeding up the calculations, and improving the efficiency of HyPS. 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. HyPS takes advantage of these layers to reduce the communication overhead of feature map transmission between V-partitions.

On a CNN structure, there are two main scenarios. First, it is possible to apply only vertical partitioning on the optional split point on the condition that

the generated V-partitions are uncomplicated and can be executed on edge device(s). Otherwise, applying vertical and horizontal partitioning is indispensable enable the partitioned model inference.

The optional split points are helpful in several use cases, when the user needs to distribute the partitioned model inference on multiple devices, and/or 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 without any bottlenecks. HyPS can be used by a simple customer or 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 easily. The model structure, in addition to edge infrastructure, should be known before starting the partitioning process.

Identifying these optional split points leads to making several combinations. In the case of the VGG16 structure, there are five strategic split points (five pooling layers), so there are several possible V-partitions. The pooling layers throughout the VGG16 model have different output dimensions. The pooling layers' output dimensions decrease when going closer to the final output layer. For example, the VGG16 output feature map dimension passes from 112*112 in the first pooling layer to 7*7 in the last. Theoretically, splitting the model on the last strategic point, with the most miniature feature map, is more efficient in minimizing communication overhead.

In this work, HyPS was applied to the VGG16 model to perform the model inference on Raspberry PI 3 B+. The VGG16 model requires horizontal partitioning on the first fully connected layer (fc6), which is very complex and cannot be executed on the targeted infrastructure. Figure 4 shows the VGG16 structure with the positions of the mandatory partitioning locations represented by continuous red lines and optional partitioning locations represented by dashed green lines. Mandatory split points are located before and after the first fully connected layer. So, VGG16 must be partitioned vertically on this position, and the fc6 must be split horizontally. The optional split points bring the opportunity to get more than three V-partitions, while minimizing the communication overhead.

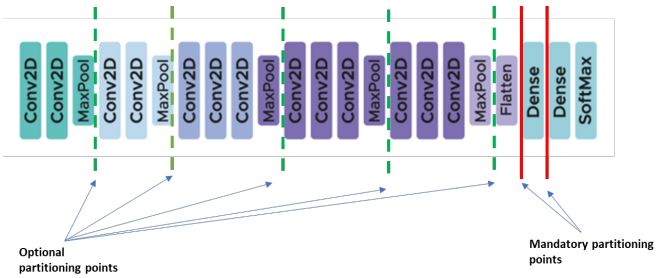

Fig. 4: Mandatory and optional split positions of VGG16 model structure.

The generated partitions can be deployed separately and distributed either over time, or spatially into multiple devices. The way how partitions are executed is known as scheduling. Figure 5 reveals only one distribution scenario among several, to illustrate the proposed contribution well. This figure reveals only one distribution scenario among several to illustrate the proposed contribution well. The cluster is composed of four edge devices. In this schema, T1, T2, T3, T4, and T5 are consecutive partitions' instantiation and execution times. In this case, the VGG16 is partitioned into four V-partitions 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 specific edge device, as discussed further in the evaluation part of this article.

The inference of the partitioned VGG16 is launched following a sequential process. The V-partitions are executed one after the other. Only the H-partitions of the fc6 can be performed in parallel, as shown in Figure 5. Besides this particular scenario, advanced scheduling possibilities exist and are led to further publications. The next subsection describes HyPS architecture entities responsible for partitioning and scheduling.

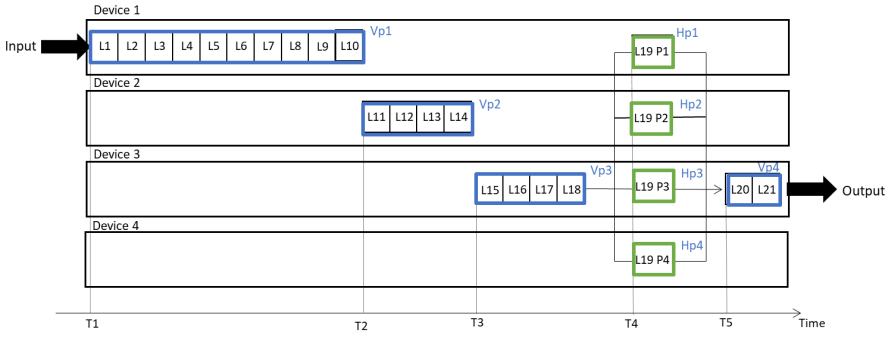

Fig. 5: hybrid partitioning of VGG16 model deployed on a cluster of four edge devices.

3.4 Architecture Overview

A particular architecture is defined to distribute inference and run partitioned model on edge infrastructure. This architecture is composed of two tree topologies: one for computations purposes and the other for related communications. Each topology is formed through the instantiation of a hierarchical star pattern at each level.

Computations Topology The computations topology comprises two types of entities: one *manager* and multiple *workers*. The manager is responsible for the

AI model partitioning and the partition distribution on device nodes. A manager can have workers that serve as sub-managers that manage a distinct group of workers. Every manager manages the execution of a given AI model through successive jobs, by scheduling the execution of partitions on different workers. The different entities constituting the computations topology can be seen in Figure 6. Figure 6 shows an example of a distributed inference of a partitioned VGG16 that is used in an industrial use case inspired by [17]. The manager is denoted "M1" while the workers are denoted "W machine". The manager decides the number of partitions taken by every worker. To simplify, only one manager and three workers are represented in Figure 6. 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 dotted lines represent the communication links between the entities.

Communications Topology The communications topology is an overlay of the computations topology. It manages data exchanges between entities. A communication hub (denoted "C1" in Figure 6) transmits data between the manager and its workers.

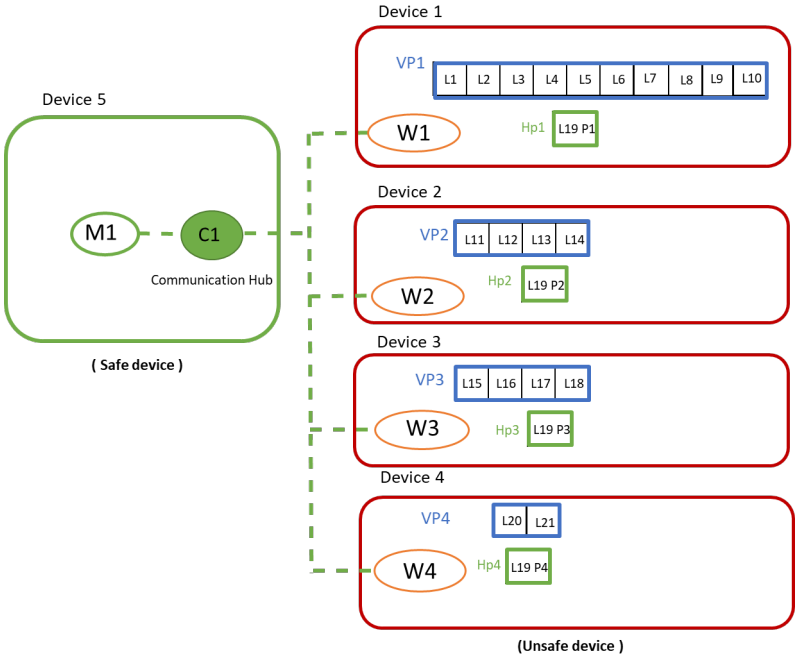

Fig. 6: Example of distributed inference architecture of VGG16 model.

The manager and the workers know how to contact the communication hub. This one separates their communications thanks to a specific addressing policy. The communication hub allows for a reactive approach for each worker, where each starts computing only when data are effectively present at a given address. For example, Worker W3 in Figure 6 does computations of V-partition 3 (Vp3) only when the feature map coming from computations of V-partition 2 done by worker 2 (Vp2) is returned to the communication hub.

The proposed architecture presents different advantages concerning edge infrastructure specificities, in particular resilience, reliability issues and also secrets protection.

Resilience & Reliability In addition to limited computing and memory resources, edge devices can suffer from many dysfunctions. Among them, energy cutoffs can occur, as is the case with customers who abruptly unplug their internet 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[18].

HyPS permits the recovery of already done computations because the communication hub ensures the persistent storage of the intermediate data exchanged between workers. Such reliable storage is fundamental in case of a problem during the inference process. The stored data can then be used as backups to resume 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. Inference results of the same partitions can be done on multiple nodes for failure detection purpose [19], thanks to a voting consensus.

Secrets protection Another concern, regarding edge infrastructure, relates to secrets protection, and more precisely, to 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. Preserving both of them is an important concern that HyPS addresses.

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 regarding secret protection. The manager and the communication hub must be protected. They must be hosted on safe devices that cope with strong security policy (represented in the green rounded corner rectangle in Figure 6). The workers are considered "unsafe" (described in the red rounded cornered rectangles in Figure 6). Regarding entities' knowledge about described secrets in the Figure 6 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, *it is possible to manage not only the possible data divulgation but also, the NN model divulgation. To the best of our knowledge, the literature weakly covers this second aspect.* 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, each manager is ensured neither to know the entire NN nor the actual data because it could be a sub-manager. Therefore, revealing the actual image after layers operations is difficult. It is demanding to interpret the partial data since having undergone several unknown transformations.

HyPS architecture and principles have been implemented in a prototype that permits a first round of experiments.

4 Implementation and Experimental Results

This section details the system setup, implementation of the proposed approach, and interpretation of experimental results that show the behavior of HyPS execution, with a real use case on a practical testbed.

4.1 System Setup

The methodology used consists in two steps. First, partitioning the model into sequential sub-networks which can then be sent to edge nodes. The partitioning function is implemented in Python, using TensorFlow and Keras libraries. Second, after the partitioning phase, the distributed inference process is executed on a realistic collaborative edge computing testbed that consists of four Raspberry Pi's: 1 Raspberry PI 3 model B (ARM Cortex-A53 Quad-Core processor - 1 GB RAM) and three Raspberry PI 3 model B+ (Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz- 1 GB RAM). All devices are connected within the same network via a LAN cable. The capabilities of those nodes represent typical edge devices [17]. According to the official website of Keras, VGG16 is a large model (528 MB) and it comprises 138.4 M of parameters. Experiments are done using the VGG16 NN model that is specified by a chain topology, and trained with the ImageNet dataset. In the following experiments, images of fixed size of 224x224 are used.

4.2 Experimental Results

In this section, experimental results are presented. It was not possible to achieve direct comparison of performances between HyPS and other existing solutions,

as it appears to be not relevant. As described further in this article, existing solutions either imply to offload computations in the cloud, or do modifications to the original NN resulting in accuracy loss. HyPS permits to process inference at the edge, without requiring offloading or NN modification. These characteristics cope with industry realistic use cases for which, strict performances must be guaranteed To evaluate the proposed strategy, a set of experiments was carried out to observe the effects of running inference of partitioned model on the inference time and the communication overhead. The following metrics are measured for each experiment:

- *Inference time:* the time necessary for the whole inference. It includes both the computation and communication time.
- *Computation time:* the duration for nodes to process an inference, excluding communications.
- *Communication time:* the transfer time of intermediate feature map from one partition to another.

Experiments are performed in two steps. The first step consists in the identification of both the mandatory and the best optional split points in this 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 and the impact on communication overhead. After choosing the best positions to split the model, the horizontal partitioning is applied in the second step to enable the execution of each complex layer. This second step is discussed further, noticeably to measure the impact of the number of H-partitions on the inference performances.

The first experimentation objective is to evaluate the impact of vertical partitioning and layer types on communication overhead and inference time.

Impact of the Vertical Partitioning on the Communication Overhead

The VGG16 model is partitioned vertically to get the smallest possible V-partitions: one partition includes only one layer. 21 V-partitions are generated since the model contains 21 successive layers. Figure 7 shows the positions of the vertical partitioning.

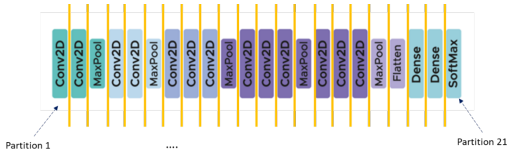

Fig. 7: VGG16 model partitioned vertically on 21 partitions.

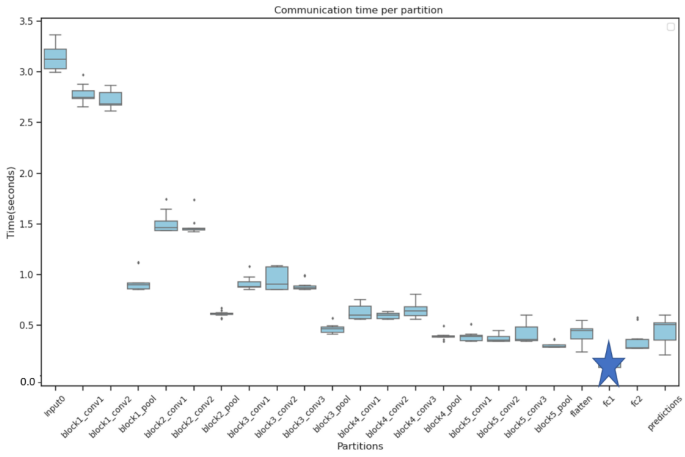

Fig. 8: Communication overhead of partitioned VGG16 model vertically on 21 partitions deployed on Raspberry Pi.

The next step is to distribute the partitioned model inference over time, and execute partitions on the testbed. The execution of the whole VGG16 is done on the Raspberry Pi with one exception: the V-partition that contains the first fully connected layer. Because this layer in too complex, it cannot be executed without a horizontal partitioning. This particular point is discussed further, but for the present experiment, the problematic V-partition is simply offloaded onto a PC. In this experimentation, the measures on the following figures related to the offloaded partition (represented by a blue star) are ignored. Only measures of the V-partitions executed on edge devices are under consideration. This experimentation allowed to precisely locate the layer that poses a problem when running VGG16.

Figure 8 presents the communication overhead for the 20 V-partitions runnable on a Raspberry Pi. The communication overhead decreases from one layer to another. This decrease is explained by the reduction of the dimension of the feature maps generated by the pooling layers. It appears that the communication overhead depends on the dimension of the feature map, the larger the feature map, the slower it is transmitted. The number of parameters does not impact the communication cost because the convolution layers with the lowest communication cost are the layers with a high number of parameters Figure 9 presents the measured feature map size generated by each V-partition. The measured size of data transmitted between V-partitions appears to be directly correlated to the feature map dimensionality and by extension, to the communication overhead measured above. This chart shows local minima in the feature map size map generated by the pooling layers.

Orange color arrows indicate these local minima in Figure 9 which correspond to pooling layers in the VGG16 structure. To minimize the communication over-

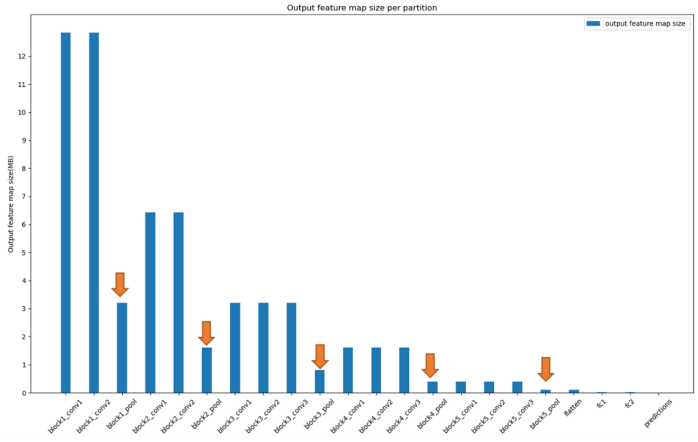

Fig. 9: Output feature map size per V-partition.

head, it is efficient to split the model after the pooling layers. This positions represented the optional strategic split points. Besides, all the pooling layers do not have the same impact on the communication overhead : the closer to the output the pooling layer, the less the communication overhead. Therefore, the optional split points that are close to the output layer should be favored.

Figure 10 shows a vertical partitioning on two different locations in the VGG16 structure: (a) presents the VGG16 model partitioned on nine partitions, the output layer of the first five partitions is a convolutional layer, (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 experimentation is to measure the communication overhead in the two cases and compare the results. The difference between convolutional and pooling layers is that 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 number of parameters and 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 11 show, respectively, the overall communication time when, the output layer in the partitions of the VGG16 are either convolutional layers or, pooling layers. The values measured for the fc6 are ignored 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 graph (a), the communication time is two times higher than the communication time in a graph (b). The difference between the cases is the size of data transmitted between partitions because the pooling layers reduce

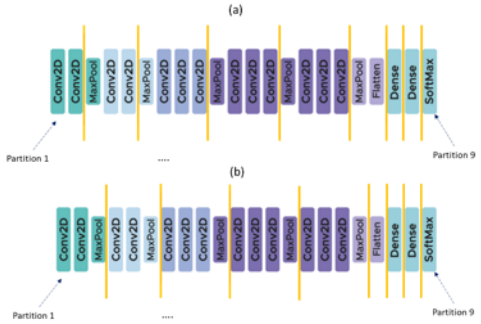

Fig. 10: (a) VGG16 model partitioned vertically on convolutional layers (b) VGG16 model partitioned vertically on pooling layers.

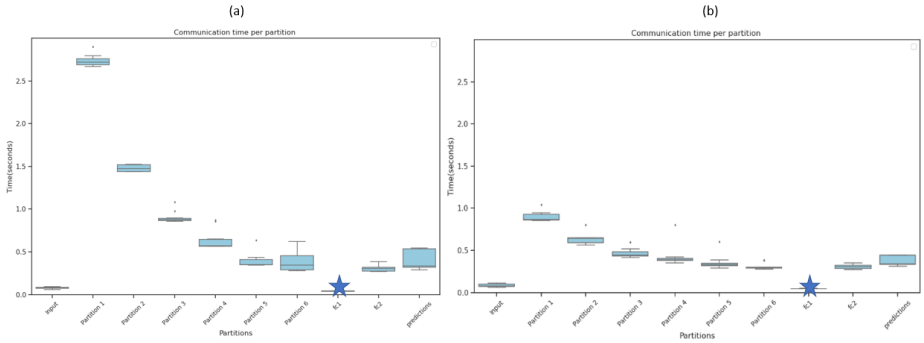

Fig. 11: (a) Communication time of partitioned VGG16 on convolutional layers (b) Communication time of partitioned VGG16 on pooling layers.

the shape of the feature map generated by the convolution layer just before. To conclude, these experiments show that:

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 the testbed**,
3. the V-partition that obstructs the inference process contains the first fully connected layer, and this is why two mandatory split points are required: after and before the fc6, 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.**

Next experiments aims determining the optimal number of H-partitions for the particular case of the VGG16 that allows fc6 inference on edge device.

Impact of the Hybrid Partitioning on the Inference Time and the Communication Overhead VGG16 is partitioned first vertically, on the mandatory split points and then, horizontally on the fc6 split point. Partitions are executed on a single Raspberry Pi using a wide range of H-partitions numbers. 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 exhaustion.

The swap activation adds more virtual memory, allowing the system to deal with more memory-intensive tasks without out-of-memory errors or, having to shut down other processes. However, the downside is that accessing the swap file significantly slows down the process and finally, increases the inference time. Swapping is not satisfactory regarding devices efficiency. That is why, for next experiments, we deactivate swap to ensure partitions are light enough.

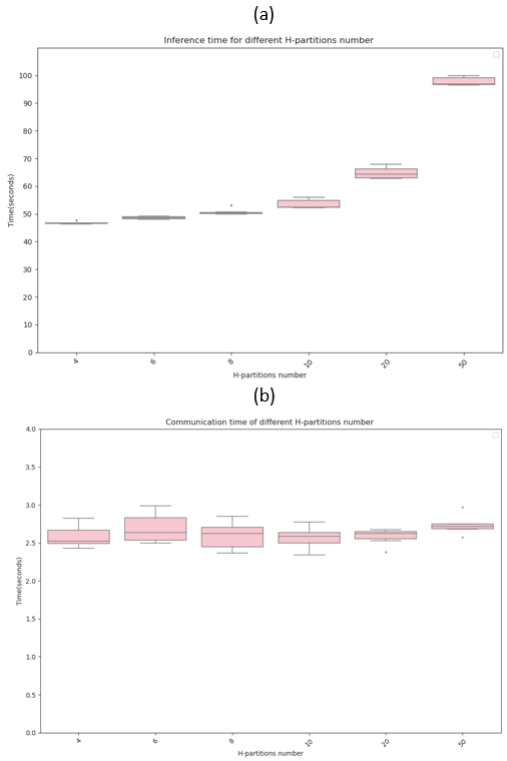

Fig. 12: (a) Inference time of partitioned VGG16 with different number of H-partitions. (b) Communication overhead of partitioned VGG16 with different number of H-partitions.

VGG16 is partitioned on the mandatory split points identified by HyPS. These points are shown in red lines in figure 4, then the complex layer fc6 is partitioned horizontally into H-partitions. According to the memory constraints, it is mandatory to split the fc6 at least into four H-partitions. With more than four H-partitions, a Raspberry Pi is able to support the inference of the entire VGG16. This number should not be too high to prevent from getting a huge communication overhead.

The next step is to try a different number of H-partitions and to observe if it impacts the inference time and the communication overhead. Figure 12 shows the results of testing VGG16 inference on a single device with different H-partitions numbers. The box plots in graph (a) 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 fc6 into 4 H-partitions. The graph (b) in Figure 12 presents the communication overhead for different numbers of H-partitions. To minimize the communication overhead, four H-partitions is the optimal number.

These experimental and practical results ,based on the implemented prototype, validate the proposed approach. Indeed, horizontal partitioning leads to high communication 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 hybrid partitioning strategy**.

5 Related Work

This section discusses a variety of related approaches which perform DNN inference on resource-constrained edge devices. Existing solutions are classified into two categories to address this issue. First, many works reduce the model size using specific model compression techniques that modify the model architecture, and make it lighter by reducing model parameters. There are four main categories of model compression techniques: quantization [20] [21], pruning [22], Knowledge Distillation(KD) [23], and low-rank factorization [24]. Model Compression broadly reduces model size and latency overhead, and allows DNN inference at the edge. However, these techniques reduce accuracy for large and complex models, and require retraining to recover the model performance. The proposed solution avoids the retraining step, and provides an inference response with the original model accuracy.

Second, several approaches adopt different partitioning strategies to split DNN structure into small partitions that can be distributed and deployed separately on IoT devices. A lot of research consider DNN structure as a graph, and use graph partitioning techniques to split the NN [25] [26][27]. Some researches focus on DNNs with chain topology, and apply partitioning strategy according to the characteristics of NN structure.

In [28], *Zhao et al.* 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. This approach allows to minimize the RAM memory footprint by reducing the sizes of input and output activations. However, its main drawback is the replication of the network's parameters on all the devices, thus increasing the memory footprint at system level.The CNN layers with huge number of parameters (e.g., fully connected layers) are not partitioned, and they are deployed on powerful gateway device. In contrast, our methodology focuses on partitioning large CNNs, and deploys all model partitions on resource-constrained devices.

In [29], *Mao et al.* 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. When the number of workers increases, the latency of MoDNN increases rather than decreases. Due to the frequent synchronization, MoDNN is sensitive to the network environment. In contrast, horizontal partitioning is restricted as much as possible to a single layer. Moreover, increasing the number of workers reduces the inference time, thus reducing latency.

Vertical partitioning has been used in many previous works. For example, *Tang et al.* in [30] proposes a vertical partitioning strategy to split the CNN model and perform inference at the edge. This work aims to reduce the memory requirement per edge device. The algorithm used to do vertical partitioning is not suitable for complex problems because it requires more significant amounts of computing power and extremely complex fitness models. Also, this algorithm is computationally expensive and time-consuming. On large CNNs, more than vertical partitioning is needed to obtain inference response on the edge device. The advantage of HyPS is the simplicity of the partitioning strategy, and the coupling of both vertical and horizontal partitioning.

In the same direction, *Kang et al.* in [31] propose to partition a DNN model vertically, and distribute partitions between cloud server and mobile device, according to the network situation. Neurosurgeon is one of the first works to investigate layer-wise partitioning. The split point is decided intelligently depending on the infrastructure network conditions and devices capacities.

In this work, partitioning at the layer granularity can provide significant latency and energy efficiency improvements. Partitioning between the last pooling layer (pool5), and the first fully connected layer (fc6 in Fig 1) of the Alex Net architecture, achieves the lowest latency. In our work, we split the DNN model systematically based on the layer's type, and we distribute partitions across only edge devices without any recourse to cloud servers. Authors in [32] 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.

F.Xue et al. in [33] proposed a locally distributed DNN inference framework based on layer-wise and fused-layer parallelization. EdgeLD can dynamically and flexibly partition a DNN model for parallel execution, to adapt to heterogeneous computing resources and different network conditions. However, our strategy enables sequential inference execution and aims to reduce communication costs.

6 Conclusion and Future Works

This article proposes a hybrid partitioning strategy that performs partitioning of large CNNs thanks to the identification of the best split points. Large CNNs can successfully be run on the resource-constrained device(s), while minimizing inference time and communication overhead.

The main contributions of this paper are: i) a hybrid partitioning strategy for running large CNN model inference on resource-constrained edge devices including a method for identifying mandatory and optional split positions in a CNN structure. ii) an architecture and a prototype (called HyPS) that implements the proposed approach, and iii) first experiments and evaluation of HyPS on a realistic testbed concerning a typical CNN.

Experimental results show that: i) the split point position impacts communication overhead, ii) the number of partitions from a horizontal partitioning influences the overall communication overhead, and iii) partitions can be scheduled sequentially on a single device or distributed on multiple devices. Experiments highlight that HyPS helps to choose the right partitioning, generates partitions ready to be deployed separately at the edge, and schedules subsequent tasks execution.

This work opens multiple perspectives. First, many other experiments have been left for the future. For instance, it would be interesting to analyze the experiment results of running a partitioned model on multiple devices. Another promising direction could investigate *batch inference*, which would generate predictions on a batch of observations on single and multiple devices.

References

1. Raza, Muhammad Raheel, Asaf Varol, and Nurhayat Varol. "Cloud and fog computing: A survey to the concept and challenges." 2020 8th International Symposium on Digital Forensics and Security (ISDFS). IEEE, 2020.
2. Abdalla, Peshraw Ahmed, and Asaf Varol. "Advantages to disadvantages of cloud computing for small-sized business." 2019 7th International Symposium on Digital Forensics and Security (ISDFS). IEEE, 2019.
3. Krutz, Ronald L., Ronald L. Krutz, and Russell Dean Vines Russell Dean Vines. Cloud security a comprehensive guide to secure cloud computing. Wiley, 2010.
4. Venugopal, Srikumar, et al. "Shadow puppets: Cloud-level accurate AI inference at the speed and economy of edge." USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18). 2018.

5. Deng, Shuiguang, et al. "Edge intelligence: The confluence of edge computing and artificial intelligence." IEEE Internet of Things Journal 7.8 (2020): 7457-7469.
6. Ademola, Olutosin Ajibola, Mairo Leier, and Eduard Petlenkov. "Evaluation of Deep Neural Network Compression Methods for Edge Devices Using Weighted Score-Based Ranking Scheme." Sensors 21.22 (2021): 7529.
7. Berthelier, Anthony, et al. "Deep model compression and architecture optimization for embedded systems: A survey." Journal of Signal Processing Systems 93.8 (2021): 863-878.
8. Kang, Yiping, et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge." ACM SIGARCH Computer Architecture News 45.1 (2017): 615-629.
9. Zhang, Shuai, et al. "Deepslicing: collaborative and adaptive cnn inference with low latency." IEEE Transactions on Parallel and Distributed Systems 32.9 (2021): 2175-2187.
10. Kim, Jung Hwan, Alwin Poulose, and Dong Seog Han. "The Customized Visual Geometry Group Deep Learning Architecture for Facial Emotion Recognition." Available at SSRN 4087604.
11. Russakovsky, Olga & Deng, Jia & Su, Hao & Krause, Jonathan & Satheesh, Sanjeev & Ma, Sean & Huang, Zhiheng & Karpathy, Andrej & Khosla, Aditya &Bernstein, Michael &Berg, Alexander & Fei-Fei, Li. (2014). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision. 115. 10.1007/s11263-015-0816-y.
12. Althubiti, Sara A., et al. "Circuit Manufacturing Defect Detection Using VGG16 Convolutional Neural Networks." Wireless Communications and Mobile Computing 2022 (2022).
13. Khanum, Abida, Chao-Yang Lee, and Chu-Sing Yang. "Deep-Learning-Based Network for Lane Following in Autonomous Vehicles." Electronics 11.19 (2022): 3084.
14. Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," Proc. IEEE, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.
15. Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.
16. Strubell, Emma & Ganesh, Ananya & Mccallum, Andrew. (2019). Energy and Policy Considerations for Deep Learning in NLP. 3645-3650. 10.18653/v1/P19-1355.
17. L. Letondeur, F.-G. Ottogalli and T. Coupaye, "A demo of application lifecycle management for IoT collaborative neighborhood in the Fog: Practical experiments and lessons learned around docker," 2017 IEEE Fog World Congress (FWC), 2017, pp. 1-6, doi: 10.1109/FWC.2017.8368526.
18. Umar Ozeer, Loïc Letondeur, Gwen Salaün, François-Gaël Ottogalli, Jean-Marc Vincent, F3ARIoT: A framework for autonomic resilience of IoT applications in the Fog, Internet of Things, Volume 12,2020, 100275,ISSN 2542-6605.
19. Chaurasia, Bhavana, and Anshul Verma. "A comprehensive study on failure detectors of distributed systems." Journal of Scientific Research 64.2 (2020).
20. Gholami, Amir, et al. "A survey of quantization methods for efficient neural network inference." arXiv preprint arXiv:2103.13630 (2021).
21. Garifulla, Mukhammed, et al. "A Case Study of Quantizing Convolutional Neural Networks for Fast Disease Diagnosis on Portable Medical Devices." Sensors 22.1 (2021): 219.
22. Lin, Shaohui, et al. "Towards optimal structured cnn pruning via generative adversarial learning." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019.

23. Wang, Guo-Hua, Yifan Ge, and Jianxin Wu. "Distilling knowledge by mimicking features." IEEE Transactions on Pattern Analysis and Machine Intelligence (2021).
24. Noach, Matan Ben, and Yoav Goldberg. "Compressing pre-trained language models by matrix decomposition." 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.
25. Zhang, Beibei, et al. "Dynamic DNN Decomposition for Lossless Synergistic Inference." 2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2021.
26. Hu, Chenghao, and Baochun Li. "Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices." IEEE INFOCOM 2022-IEEE Conference on Computer Communications. IEEE, 2022.
27. Jeong, Hyuk-Jin, et al. "IONN: Incremental offloading of neural network computations from mobile devices to edge servers." Proceedings of the ACM Symposium on Cloud Computing. 2018.
28. Zhao, Zhuoran & Barijough, Kamyar & Gerstlauer, Andreas. (2018). DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
29. J. Mao, X. Chen, K. W. Nixon, C. Krieger and Y. Chen, "MoDNN: Local distributed mobile computing system for Deep Neural Network," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1396-1401, doi: 10.23919/DATE.2017.7927211.
30. Erqian Tang and Todor Stefanov. 2021. Low-memory and high-performance CNN inference on distributed systems at the edge. Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion. Association for Computing Machinery, New York, NY, USA, Article 26, 1–8. https://doi.org/10.1145/3492323.3495629
31. Kang, Yiping & Hauswald, Johann & Gao, Cao & Rovinski, Austin & Mudge, Trevor & Mars, Jason & Tang, Lingjia. (2017). Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. ACM SIGARCH Computer Architecture News. 45. 615-629. 10.1145/3093337.3037698.
32. Mehta, Rishabh, and Rajeev Shorey. "Deepsplit: Dynamic splitting of collaborative edge-cloud convolutional neural networks." 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS). IEEE, 2020.
33. F. Xue, W. Fang, W. Xu, Q. Wang, X. Ma, and Y. Ding, "EdgeLD: Locally distributed deep learning inference on edge device clusters," in Proc. IEEE 22nd Int. Conf. High Perform. Comput. Communications; IEEE 18th Int. Conf. Smart City; IEEE 6th Int. Conf.Data Sci. Syst. (HPCC/SmartCity/DSS), Dec. 2020, pp. 613–619, doi: 10.1109/HPCC-SmartCity DSS50907.2020.00078.

3