Distributed Inference with Early Exit at Edge Networks

BY

MARCO COLOCRESE
B.S., Politecnico di Torino, Turin, Italy, 2021

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2023

Chicago, Illinois

Defense Committee:

Hulya Seferoglu, Chair and Advisor
Erdem Koyuncu
Enrico Magli, Politecnico di Torino

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CNN	Convolutional Neural Network
CV	Computer Vision
DC	Distributed Computing
DNN	Deep Neural Network
EE	Early Exit
FI	Federated Inference
FL	Federated Learning
FLOP	FLoating Point Operation
IoT	Internet of Things.
IoV	Internet of Vehicles.
PCA	Principal Component Analysis
PO	Partial Offloading
RNN	Recurrent Neural Networks.
SC	Split Computing.
UIC	University of Illinois at Chicago

SUMMARY

With the increasing prevalence of edge devices and the exponential growth of deep learning applications, there is a pressing need for efficient algorithms and techniques that can be applied to resource-constrained devices. This master thesis presents a novel system that combines distributed computing and early exit strategies to enable deep learning on edge devices. A multi-threaded algorithm is proposed to flexibly manage the load on each device based on both communication and computational requirements. Two solutions are presented, addressing common needs: accuracy constraint and input rate constraint.

The primary objective is to investigate the feasibility, performance, and flexibility of the proposed techniques in resource-constrained environments. The evaluation of the framework includes performance benchmarking, analysis of different neural network architectures and network topologies, and assessment of its adaptability. The results demonstrate effective resource exploitation, showcasing superior performance in topologies consisting solely of edge devices compared to traditional approaches or partial offloading to an edge server. The findings of this study highlight the potential of the proposed approach in enhancing deep learning capabilities on resource-constrained edge devices, particularly in server-less topologies.

CHAPTER 1

INTRODUCTION

1.1 Deep Learning at the Edge

Over the past ten years, there has been significant progress in improving deep learning models, which have been increasingly utilized in a wide range of applications. Moreover, there has been a growing focus on edge computing and subsequently the utilization of deep learning on edge devices. In this section, we will provide examples of deep learning applications at the edge to illustrate its growing significance.

1.1.1 Computer Vision

Computer Vision (CV) is a field that focuses on enabling computers to interpret visual information from digital images or videos, allowing them to recognize objects, detect patterns, track motion, and perform tasks like object classification, image segmentation, face recognition, and scene understanding. In edge computing, real-time video analysis and image recognition are key applications of CV.

Real-time video analysis systems are widely used in various applications, such as drones and traffic surveillance. Wang et al. [1] presented an architecture for video analytics on small autonomous drones that utilizes edge computing to optimize bandwidth usage. This enables real-time video analytics. Ali et al. [2] devised a deep learning-based video analytics system that operates at the edge, specifically designed to identify objects in large-scale IoT video

streams. The system comprises various stages, including frame loading and decoding, motion detection, preprocessing, object detection and recognition. The first three stages are executed on the edge, while the last stage takes place in the cloud. Kar et al. [3] developed a CNN (Convolutional Neural Network) system capable of analysing video to compute the number of vehicles estimating traffic state. Their model was deployed on a dashboard camera onboard a vehicle which detects other vehicles on the road.

Image recognition in deep learning involves training computers to identify and classify objects or patterns in digital images. It has shown remarkable performance in applications such as object detection, image classification, facial recognition, and image segmentation. It has transformed fields like autonomous driving, medical imaging, and content-based image retrieval by enabling computers to accurately understand visual information.

The rise of cameras on mobile and IoT devices give more and more importance to techniques devoted to image recognition. For example, in wildlife monitoring, the presence of such devices has created a need for advanced image recognition capabilities. Additionally, processing data on the edge has become important for devices with limited bandwidth. By processing photos and videos on the edge, the data can be adjusted to an appropriate resolution before being uploaded to the Internet. Lightweight frameworks like Caffe2Go [4] have been developed to deploy deep learning systems on mobile devices, reducing the input layer size of the models.

In the context of image recognition, Liu et al. [5] introduced a food recognition system that operates in real-time, leveraging neural networks techniques on the edge computing scenario. The system processes images at the edge and utilizes a cloud-based Convolutional Neural Net-

work (CNN) for classification which is exploited to compute the amount of assumed calories. The edge device's main function is to enhance the quality of blurry food images taken by the user before sending them to the cloud server for further processing.

1.1.2 Speech Recognition

Deep learning-based speech recognition involves using advanced neural network models to convert spoken language into written text. Models such as Recurrent Neural Networks (RNNs) and transformers are trained on large datasets of spoken language to understand the patterns and connections between audio input and corresponding textual output. These models capture acoustic features and language context to accurately transcribe speech. The usage of deep learning models for speech recognition on devices with limited computational capabilities can be challenging. However, techniques like EdgeSpeechNets [6] have been introduced to enable efficient deployment on mobile edge devices. These models utilize deep neural networks to extract features, generate posterior probabilities, and produce output scores for recognizing audio.

1.1.3 Autonomous vehicles

Autonomous vehicles generate a substantial amount of data, averaging over 4 TB every 90 minutes of driving [7], which requires real-time processing for driving decisions. Due to limited bandwidth, transferring such enormous data to remote servers is impractical. Therefore, edge computing plays a crucial role in autonomous driving systems. Chen et al. [8] proposed a cognitive internet of vehicles design, focusing on networks composed of vehicles. Liang et al. [9] employed reinforcement learning to manage network resources in vehicular networks.

Additionally, DeepCrash [10] is a an IoV system designed to detect and communicate collision events in hsort time, which exploits both edge computing and cloud offloading.

1.1.4 Smart homes and Safety

There is an increasing trend of designing homes with intelligent systems that utilize resource-constrained devices. Ensuring the safety of children and elderly individuals is an important aspect of smart homes: Hsu et al. [11] designed a fall detection system that utilizes skeleton extraction and machine learning prediction models for detection, enhanced by video and image compression using a Raspberry Pi. Computations are performed on the cloud, then notification are generated accordingly.

Miraftabzadeh et al. [12] introduced a neural network technique for enhancing pedestrian security. This technique is thought to allow people identification in overcrowded environments. In their approach, edge devices initially detect the face and compare it with embeddings stored in the device itself. If no results arises, data are sent to cloud to perform new comparisons and store it.

Liu et al. [13] proposed a system that operates at the edge in the context of ride-sharing to detect attacks and events which can lead to harmful outcomes. Their system combines audio, video, and driving behavior data for comprehensive analysis. The smartphones of both drivers and passengers act as edge devices, triggering the transmission of video data to the cloud. A trained CNN model analyzes the video data in the cloud. To reduce the bandwidth required for cloud upload, data dimensionality can be reduced at the edge through compression techniques.

1.2 Motivation of the study

Given the growing significance of deep learning at the edge, as discussed in the preceding section, it is imperative to develop systems capable of effectively managing cooperative environments comprising resource-constrained devices. Existing literature predominantly focuses on static hierarchical systems with predefined paths for inference, assuming prior knowledge of device states and capabilities and of the topology they represent. However, our system takes a more generalized approach. It can adapt to systems that include diverse edge devices with varying capabilities, without the need for prior knowledge about the devices involved and independently from the system topology and the specific Deep Learning application. This flexibility allows for more dynamic and efficient utilization of available resources.

We propose a novel system that introduces a generalized approach to combining Early Exit (EE) and Distributed Computing (DC). This approach can be applied to various systems composed of different devices, regardless of their capabilities. The two keys elements are:

- **Distributed Inference:** Edge devices often have limited resources in terms of memory and computational capabilities. Distributed inference aims to address this limitation by enabling cooperation among edge devices. By parallelizing inference and leveraging the capabilities of different devices, the overall workload can be distributed more effectively.
- **Early Exit:** To reduce inference latency, it is crucial to consider the fixed computational capacity of each device, measured in terms of FLOPs per second. One approach to mitigate this challenge is to reduce the number of FLOPs required for inference. This is where early exit comes into play. It allows for intelligent decision-making, enabling the

system to bypass unnecessary computations by exiting early when the input is deemed easy enough for the given task (such as classification in this work). This tradeoff between latency and accuracy is carefully balanced.

1.3 Background

In this section, we will give an intuition about the main concepts we exploited for this work, which will be explored in more details in Chapter 2.

1.3.1 Early Exit

As in most deep learning applications, in the context of computer vision tasks, specifically classification, it is important to acknowledge that not all inputs exhibit the same level of difficulty, which pertains to the complexity of their respective classifications. When designing and training a Deep Neural Network (DNN), the primary objective is to maximize accuracy, thereby creating systems capable of accurately classifying more complex inputs. However, even within the same dataset, certain inputs can be easily classified with fewer computations. This concept underlies the notion of "Network Overthinking" introduced by Kaya et al. in [14]. The authors propose that a network engages in overthinking when a previous layer's representation (requiring fewer FLOPs and resulting in lower inference latency) is sufficient for accurate classification. To exploit this property of certain inputs, early exit can be used, reducing inference delay. The early exit mechanism, firstly introduced by Andres Rodriguez et al. [15], in DNNs allows for early termination of inference based on intermediate predictions, providing a trade-off between accuracy and computation cost. It enables DNN models to make predictions at multiple points during the computation process, allowing for early classification

if a confident prediction is obtained. This approach is particularly useful in scenarios where computational resources are limited or real-time processing is required.

An intuition of early exit mechanism is reported in Figure 1. In this example, a confidence level is considered in the early exit policy: the architecture is able to classify the outputs in different parts with different levels of confidence.

Figure 1: Intuition of the early exit mechanism.

1.3.2 Distributed Computing

Distributed computing in edge devices refers to the practice of performing computing tasks and data processing across multiple interconnected edge devices, such as smartphones, IoT

devices, and edge servers. Instead of relying solely on a centralized cloud infrastructure, distributed computing leverages the computational capabilities of edge devices to distribute and process data closer to the source or at the network edge. This approach offers several advantages, including reduced latency, improved scalability, increased privacy, and efficient utilization of local resources. By distributing computing tasks across edge devices, distributed computing enables faster data analysis, real-time decision-making, and enhanced performance for edge-based applications and services.

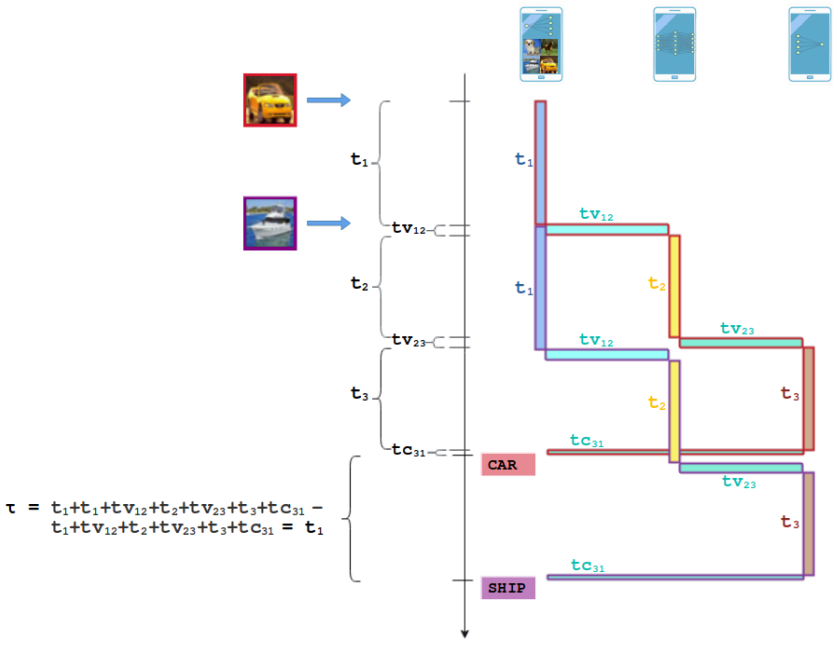

Figure 2: Parallelization in distributed computing with partial offloading.

To have an intuition of the advantages of distributed computing, let's consider a scenario where three subnetworks are assigned to three different devices for simplicity, as in Figure 2. In this case, t_i represents the time required for subnetwork inference on one input at device i, tv_{ij} represents the communication time for transmitting an activation vector from device i to device j and tc_{ij} represents the communication time for transmitting a classification output from device i to device j. We assume that the output needs to be transmitted back to the device that owns the original input.

By leveraging distributed computing, the system can distribute the workload across multiple devices and utilize parallel processing capabilities, thereby improving overall efficiency and reducing inference time. In particular, when the input queue is not empty, the computational power can be efficiently utilized by dedicating it to the initial part of the inference for another input as soon as the first device completes the inference on the first subnetwork. This allows for parallelization of the inference process among devices, resulting in a reduction of the time interval (τ) between two outputs. In the example depicted in Figure 2, the first device serves as the bottleneck since $t_1 > t_2$, $t_1 > t_3$, and it is also greater than the communication times. In a more general scenario, τ corresponds to the bottleneck time, which is the longest duration among all the considered times (communication and computation), potentially leading to the formation of a queue on the device hosting the process that requires this time. It is important to note that in this particular scenario, we have assumed that the devices are capable of simultaneously handling both communication and inference tasks independently.

1.3.3 Partial offloading: DNN-partitioning and dimensionality reduction

In literature, solutions considering the entire offloading of inputs to server has been analyzed as a suitable solution when handling resource-constrained devices. However, complete offloading of the entire DNN to the server is not the sole viable approach. On the contrary, initiating the inference at the edge and transferring intermediate activation vectors has proven to be effective, offering advantages in terms of both privacy preservation (by transmitting activation vectors instead of raw data) and optimal utilization of edge resources, thus we refer to Partial Offloading (PO), as only one portion of the computation is offloaded.

Partitioning the DNN, which refers to the division of the model between the edge device and the server/other devices, plays a critical role in this scenarios. It holds significant importance in determining how the computational workload is distributed and allocated between these entities. It basically consists in dividing the model in two parts: the Head Model (running on the mobile device) and the Tail Model (in a general scenario running on the server).

Offloading becomes crucial in wireless communication scenarios where data transmission can be the limiting factor for offloaded systems. This is especially true when dealing with high-dimensional input data. As a result, the need for dimensionality reduction arises as a solution to this challenge. In the context of neural networks, several common techniques are widely used for dimensionality reduction, aiming to reduce the dimensionality of the input data while preserving its essential information. Fully connected and convolutional autoencoders are especially well-suited for this purpose: the purpose of the encoder is to compress the activation vector by reducing its dimensionality, allowing for the transmission of smaller data and minimizing

communication delays; subsequently, the decoder is employed to restore the encoded data back to its original shape, enabling the continuation of inference using the remaining portion of the neural network. A visual representation of this mechanism is illustrated in Figure 3.

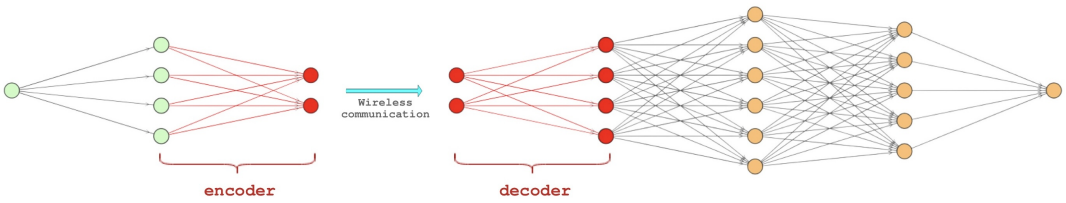

Figure 3: Fully connected autoencoder applied to split computing.

1.4 Thesis structure

This document is organized as follows:

- Chapter 2 (**Related works**): In this chapter, we present a thorough review of the relevant literature pertaining to our study. We not only explore systems that employ similar techniques but also consider alternative approaches that influenced our decision-making process. This comprehensive overview encompasses a wide range of possibilities, providing valuable insights into the rationale behind our choices. We will introduce the models used in our research, including other lightweight models suitable for resource-constrained devices. Furthermore, we will explore the techniques employed, such as Offloading, Dis-

tributed Computing, and dimensionality reduction through autoencoders. Additionally, we will introduce existing literature that describes systems similar to ours.

- Chapter 3 (**Methodology**): We introduce our multi-threaded algorithm, providing comprehensive details on the management of early exit decisions and the effective handling of communicational and computational loads. We delve into the intricacies of our algorithm, highlighting its ability to make informed decisions regarding early exit strategies. Additionally, we discuss how our algorithm efficiently manages the communication and computational loads, ensuring optimal performance and resource utilization.
- Chapter 4 (**Experimental setup and implementation**): We will delve into the details of the two models utilized, namely MobileNetv2 and ResNet50. We will explain the modifications we made to adapt them to our study. Additionally, we will provide a thorough description of our system, with a particular focus on constructing the distributed environment and on the tested topologies. Then, we will provide an in-depth description of our implementation and an outline the physical setup utilized for our experiments.
- Chapter 5 (**Results and analysis**): This chapter will present the results obtained from all the implemented scenarios. We will compare the performance of the two models and our generalized system to state-of-the-art hierarchical approaches that incorporate the usage of edge servers. Furthermore, a detailed analysis of the results will be provided.
- Chapter 6 (**Conclusion**): Finally, we will summarize our work and the results obtained. We will also discuss potential avenues for future research and enhancements.

CHAPTER 2

RELATED WORK

Many researches aim at investigating techniques for mitigating inference latency on edge devices, with a particular focus on applications commonly found in resource-constrained devices such as mobile phones, IoT devices, drones, autonomous vehicles, and network elements. Numerous studies have been conducted in this area, with the objective of enhancing the performance of edge inference latency, as well as edge servers. The results obtained from these studies have demonstrated notable improvements across various ecosystems.

In this chapter, we will give an overview of these techniques, dividing them in three main categories: **Local Computing** (which is applied to a single device) **Edge or Cloud Server Offloading** (which totally relies the computational capabilities of a server) and **Split Computing** (SC) (which exploits the distribution of the computation over different devices), an intuition of these techniques is reported in Figure 4. More advanced techniques/systems are obtained as combination of these three.

2.1 Local computing

In the context of limited computational capabilities, such as those found in local computing environments, the necessity arises to restrict the number of parameters and required operations (FLOPs) in neural network models. This requirement has prompted the development of novel models explicitly designed for resource-constrained devices, we will refer to it as **light**

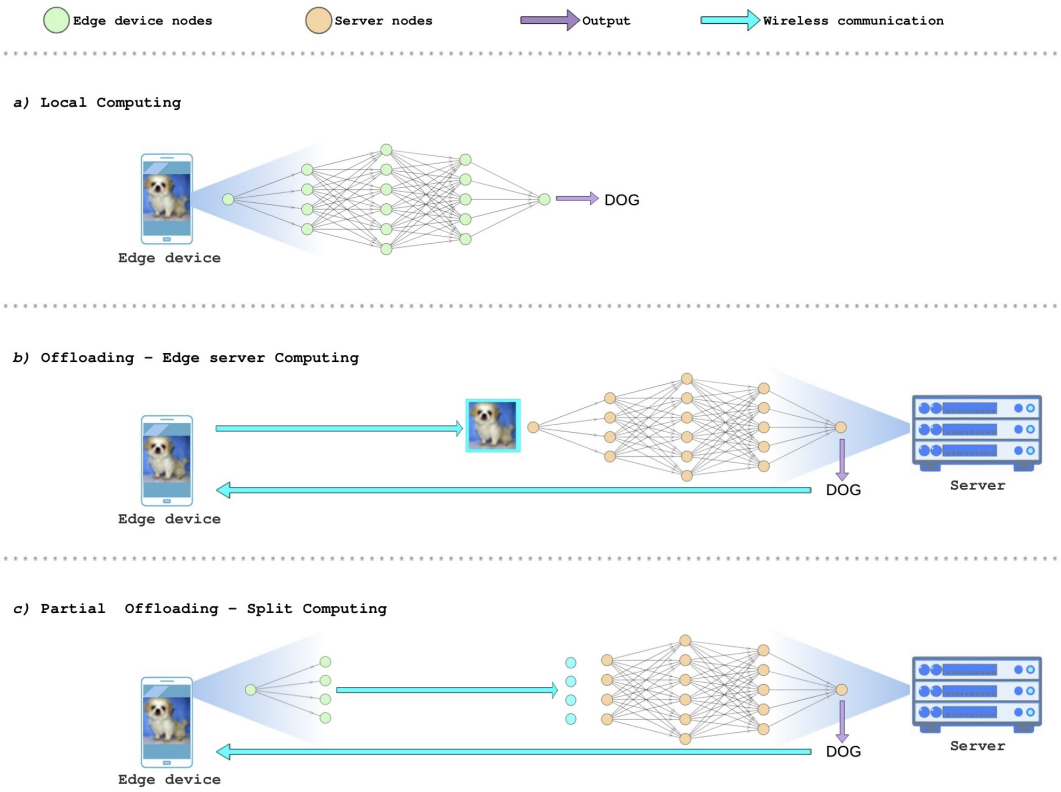

Figure 4: Intuition of local computing, server offloading and split computing.

DNN design. Additionally, techniques focusing on compressing existing DNN models, (**DNN compression**), have been employed to address this challenge.

2.1.1 Light DNN design

When designing DNN specifically for resource-constrained devices, the key idea is to find a trade-off between a good accuracy and limited memory requirement (small number of parameters) and inference latency. In this section we will cite some examples of these DNN, whose key aspects for reducing resource usage have to be find in their design: MobileNet [16], ShuffleNet [17], SqueezeNet [18], EfficientNet [19], YOLO [20], TinyNAS [21].

MobileNet is a family of lightweight DNN models specifically designed for mobile and embedded devices. These models employ depth-wise separable convolutions to reduce computational complexity while maintaining reasonable accuracy. MobileNetv2 [22] uses Inverted Residuals and Linear Bottlenecks, its architecture will be deeply analyzed in section 4.1.1.2. Thanks to its design, it reaches an accuracy of 72% on ImageNet, which is only 5% than the accuracy reached by ResNet101 [23], with a significant difference in terms of the number of parameters (3.4 million vs 44.6 million) and FLOPs required for inference (300 million vs 7.8 billion).

ShuffleNet is another family of efficient DNN models designed for mobile and embedded devices. These architectures utilize two new operations, pointwise group convolution and channel shuffle, to reduce parameter count and computational complexity while maintaining accuracy under the constraint of 40 million FLOPs.

SqueezeNet employs various techniques such as 1x1 convolutions, fire modules, and aggressive downsampling to minimize the number of parameters and operations. In its basic

implementations (without model compression), authors claimed to reach the same accuracy of AlexNet [24] on ImageNet using 50x less parameters (1.25 million).

The EfficientNet family employ a compound scaling method that uniformly reduces depth, width, and resolution to reduce computational demands. EfficientNet models offer a range of model sizes depending on the compound coefficient, allowing adaptation to different resource constraints. For instance, EfficientNet-B3 was able to reach the same accuracy of ResNet-152, while requiring x7.6 less parameters and x16 less FLOPs for inference.

YOLO is a real-time object detection model that offers a trade-off between accuracy and speed; YOLOv7 [25] can be considered as the state-of-the-art for computer vision tasks. It provides a single-pass approach for object detection, making it computationally efficient. YOLOv7 is claimed to perform better all others object detectors in both speed and accuracy in the range from 5 FPS to 160 FPS, having the highest accuracy (56.8%).

Also Reinforcement Learning models have been proposed for resource constraint devices. An example is TinyNAS: a Neural Architecture Search approach specifically targeting small models, which utilizes reinforcement learning techniques to automatically discover compact network architectures with optimized trade-offs between size, accuracy, and computational requirements.

2.1.2 Early exit

In the pursuit of reducing the burden on a single device, early exit has demonstrated its ability to decrease the average number of FLOPs needed for inference. This achievement is realized by effectively solving certain tasks without the utilization of all layers, as explained in 1.3.1. We will now cite some examples of how early exit can be implemented.

BranchyNet [15] proposes a framework that enables DNN models to dynamically branch and early exit during inference based on confidence thresholds. It demonstrates the effectiveness of the early exit mechanism in reducing computation time while maintaining competitive accuracy on various image classification tasks. Authors applied this mechanism to classification task on CIFAR-10, obtaining speedup gains of 1.5-2.4x over AlexNet and 1.9x over ResNet, without significant accuracy loss, thanks to the simplicity of the used dataset. In BranchyNet, the confidence which defines the choice of exiting or not at a specific branch, is defined by the entropy the output of the last layer before the branch (eq. Equation 2.1).

$$entropy(\mathbf{y}) = \sum_{c \in C} y_c \; log \; y_c \tag{2.1}$$

The one used in BranchyNet is not the only possible way to define the confidence level of an early classification: for instance, Gormez et al. [26] propose the usage of Class Means (E^2CM): the probability of an input belonging to a specific class is obtained as the *softmax* (eq. Equation 2.2) of the normalized euclidean distances between each layer j output and the mean of all its outputs.

$$Softmax(\mathbf{x_i}) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2.2}$$

EE approaches have been extensively explored also for Natural Language Processing (NLP) tasks, which typically involve higher computational demands than CV. For instance, state-of-the-art models like BERT [27] can contain up to 355 million parameters, significantly surpassing

the parameter count of image classification models used in split computing studies, such as ResNet-152, which has 60 million parameters, thus eventually benefiting more than computer vision from these techniques. In this field, high importance is given to latency. For instance, Zhou et al. [28] propose a *patience metric* t and the early exit policy is in a cross-layer fashion. The process of PABEE (Patience-based Early Exit) works as follow: an input instance x passes through layers L_1 to L_n and a regressor R_n to make predictions. Each layer L_i is connected to an internal classifier R_i. To determine when to stop the inference early, a counter is used. For regression, if the difference between predicted values is below a threshold τ; otherwise, it is reset to 0. (the same mechanism is also applied to classification, where the counter is incremented if the predicted class remains the same as the previous layer). Thus, when the counter reaches the desired value, the output is obtained at the early exit point. This policy shows an inference speed-up ratio between 1.30x and 1.96x, with respect to the original version of BERT.

In the literature, the usage of early exit has also been explored in combination with other techniques feasible for resource-constrained devices. For instance, Gormez and Koyuncu introduced an hybrid approach based on early exit and pruning in [29], demonstrating a 50% reduction in computational cost at the expense of a 4% decrease in accuracy, conducting experiments using ResNet-56 and the CIFAR-10 dataset.

2.2 Edge server computing

Despite the significant progress made in the field of lightweight models and compression techniques, performing inference on edge devices remains a challenge, especially when real-time constraints must be satisfied. One solution consists in computation offloading to cloud server.

However, certain applications that demand quick response times or operate in environments with limited connectivity to the cloud, such as drones or military settings, may not be suitable for cloud-based solutions. To address this, the literature proposes the utilization of edge servers, which are less powerful but closer in proximity to edge devices, resulting in reduced communication delays. Edge computing, which involves performing computation and data processing at the edge of the network, has been a natural progression driven by the need to address latency, bandwidth, and privacy concerns in various applications. The integration of neural networks with edge computing has emerged as a practical approach to enable real-time and efficient inference on resource-constrained devices.

The basic approach consists in offloading all the inference to the edge server, thus the edge devices would only need to manage the communication (data sending and output reception). To enhance this strategy, an initial refinement involves minimizing data redundancy to mitigate communication delays. This is accomplished through data preprocessing techniques.

2.2.1 Resource management and Offloading

Considering that, in general, edge servers have to manage many different tasks from different devices, a key aspect in edge server computing is a proper resource management. The most effective way has been shown to be the flexibility of the DNN configuration based on resource availability, delay requirements and accuracy requirements. Zhang et al. [30] investigated this approach, creating a system able to adapt itself runtime. The authors design a framework that dynamically adjusts the processing complexity and accuracy of video analytics algorithms based on available resources and time constraints. By utilizing approximation techniques, the system

trades off accuracy for reduced computational requirements, allowing for real-time analysis of video streams at scale.

Despite significant efforts in resource management and load balancing, certain applications necessitate the deployment of multiple servers. Various systems have been proposed to address this requirement, incorporating numerous edge servers and, in some cases, even cloud servers. This is particularly relevant in scenarios where edge devices have highly limited resources, such as in IoT systems and smart city environments. As mentioned in [31], smart cities heavily rely on server usage, especially when low-power devices are involved. The literature proposes various architectures to address this, including the Cloud-Centric Architecture (which centralizes data processing and storage in the cloud), the Fog Computing Architecture (which brings computing resources closer to edge devices, including edge devices with local processing capabilities), the Hybrid Cloud-Fog Architecture (combining elements of both cloud-centric and fog computing models to offload tasks to the cloud while performing time-sensitive or resource-intensive computations at the edge), and the Hierarchical Architecture (organizing the smart city infrastructure into multiple tiers based on proximity to edge devices, with each tier responsible for specific tasks and data flowing between tiers based on processing requirements).

Nevertheless, these techniques are not only associated with the extra expenses incurred from server usage but also may not be suitable in specific scenarios, such as remote or rural areas, disaster-stricken regions, military operations, or situations involving autonomous vehicles and drones. Hence, it becomes imperative to explore systems that can operate without depending on cloud servers and, if possible, even exclude the use of edge servers.

2.2.2 Input dimensionality reduction

When dealing with systems that handle a large volume of inputs and have the capability to offload high-dimensional vectors, it becomes essential to discover methods for reducing the communication load within the system. This is important in order to minimize communication delay and cost. Here, we will explore the commonly employed techniques: PCA and Autoencoders.

Principal Component Analysis (PCA) [31], proposed more then a century ago, is a classical statistical technique that identifies the principal components in the input data, capturing the maximum variance. By projecting the data onto a lower-dimensional subspace defined by these components, PCA provides a linear transformation that maximizes the retention of variance. It is extensively utilized for dimensionality reduction tasks.

Autoencoders are neural network architectures consisting of an encoder and a decoder. They are trained to reconstruct the input data from a lower-dimensional latent space. The bottleneck layer in the middle of the autoencoder represents the compressed representation of the data. Autoencoders effectively reduce the dimensionality by learning to capture essential features while minimizing the reconstruction error. Autoencoders have been designed for various purposes, including signal denoising, speech recognition, text summarization, and sample generation using architectures like denoising autoencoders [32], sequence-to-sequence autoencoders [33], and Generative Adversarial Networks (GANs) [34].

In this work, we specifically focus on the dimensionality reduction achieved through encoding. The two well-studied models for this purpose are fully connected autoencoders and

convolutional autoencoders. **Fully connected autoencoders** [35] [36], also known as dense autoencoders, consist solely of fully connected layers. The input data is flattened and passed through multiple fully connected layers in the encoder, gradually reducing the dimensionality. The decoder consists of fully connected layers that mirror the encoder's structure, expanding the compressed representation back to the original input shape. The training process utilizes a loss function that encourages minimizing the difference between the input and the reconstructed output.

Convolutional autoencoders [37] excel at processing structured data, particularly images. They employ convolutional layers for both the encoding and decoding processes, allowing them to capture spatial relationships and local patterns within the data. In a convolutional autoencoder, the encoder component typically consists of convolutional and pooling layers, which are then followed by fully connected layers to produce a compressed representation. The decoder mirrors the encoder's structure but employs deconvolutional layers (transpose convolutional layers) to reconstruct the original input. Similar to fully connected autoencoders, the training process of convolutional autoencoders minimizes the reconstruction error through an appropriate loss function.

These dimensionality reduction techniques will result very effective even when considering split computing (Section 2.3) and Partial Offloading (Section 2.3.1).

2.3 Split Computing

2.3.1 Partial offloading

The concept of offloading has been previously explored in systems that incorporate both edge and cloud servers (as discussed in Section 2.2.1). In such scenarios, the primary objective was to offload the entire DNN to a more capable device that could perform the inference tasks that the edge device alone was unable to handle.

As mentioned in Section 2, complete offloading of the entire DNN to the server is not the sole viable approach. On the contrary, initiating the inference at the edge has proven to be effective, offering advantages in terms of both privacy preservation (by transmitting activation vectors instead of raw data) and optimal utilization of edge resources, thus we refer to Partial Offloading, as only one portion of the computation is offloaded. The first intuition of this concept, by Cuervo et al. [38] introduced the MAUI framework, which enabled offloading computationally intensive parts of mobile applications, including DNN models, to remote servers. Although it focuses on general application code offloading, it laid the foundation for the concept of selectively offloading DNN layers to servers while keeping the lighter layers on the edge device.

In the section 2.2.2, we discussed the application of autoencoders in a situation where the input is transferred to a server. These types of architectures can also be employed to encode and decrease the dimensionality of activation vectors obtained from intermediate layers, specifically in a PO scenario.

2.3.2 DNN-partitioning

To the best of our knowledge, Neurosurgeon [39] is regarded as the pioneering system that specifically addresses DNN partitioning. The research paper focuses on diverse applications within CV, speech recognition, and NLP domains. The significant contribution of Neurosurgeon lies in the development of a scheduler capable of automatically partitioning DNN computations at the granularity of individual DNN layers. Through evaluations conducted on various DNN architectures, the authors reported substantial improvements, including an average 3.1x reduction in end-to-end latency, a 59.5% decrease in energy consumption, and a 1.5x enhancement in datacenter throughput.

Regarding DNN partitioning, Edgent [40] is a notable system that introduces an innovative approach by incorporating an online tuning stage. The underlying process involves initially partitioning the employed DNN during a static offline stage. Afterwards, Edgent dynamically evaluates the existing bandwidth conditions and jointly optimizes DNN partitioning and sizing. This optimization is based on the specified timing requirement and the setup obtained during the offline phase. The primary objective is to maximize the inference accuracy while adhering to the specified latency requirement. This integrated approach allows Edgent to adaptively adjust the DNN configuration in real-time, ensuring optimal performance in dynamic network conditions.

2.3.3 Early Exit applied to Split Computing

A very interesting line of work on this field is the application of early exit techniques to split computing and distributed computing. Teerapittayanon et al. [41], extend the concept intro-

duced in BranchyNet [15] to mobile-edge-cloud computing systems. In their system (DNNN), the mobile device is assigned the smallest neural model initially. If the model's confidence for the input falls below a certain threshold, the intermediate activation vectors are sent to the edge server. Here, the computation continues using a larger DNN model with an additional exit. If the desired confidence level is still not achieved, the intermediate output is further sent to the cloud, where the largest DNN model is executed to perform inference. In contrast to the conventional approach of offloading raw sensor data for cloud processing, DDNN (Distributed Deep Neural Network) performs local processing of sensor data on end devices, achieving high accuracy. This approach significantly reduces the communication cost by more than 20 times, thanks both to the reduction of the dimension of sent messages (activation vectors instead of raw data) and to early exit which reduces the need for server usage (and data transmission).

We will now investigate successful research which jointly apply split computing and early exit to computer vision tasks, such as Image classification. SPINN [42] exploits cloud offloading and a progressive inference reduce CNN inference in different scenarios. A key element introduced in this work is the usage of an innovative scheduler that effectively optimizes the early-exit policy and CNN splitting in real-time. This capability enables the system to adapt to runtime conditions and efficiently fulfill user-defined service-level requirements. An important introduction regards the definition of the Dynamic Scheduler, which is responsible for the dynamic distribution of the offloading from edge to cloud, also defining the early exit policy. The novelty is the consideration of the server cost, in fact, this element minimizes a cost function defined by latency, throughput, server cost, device cost and accuracy. Similar to most existing

literature on the combined utilization of early exit and split computing, this work is limited in its scope as it solely focuses on a topology consisting of a single edge device and a server.

Boomerang [43] leverages DNN right-sizing and DNN partitioning techniques to achieve low-latency and accurate DNN inference. DNN right-sizing exploits the early-exit mechanism to adjust the amount of DNN computation, thereby reducing the overall runtime. DNN partitioning dynamically distributes DNN computation between IoT devices and the edge server, leveraging hybrid computing resources for immediate inference. By combining these techniques, Boomerang selects the partition and exit points to maximize performance while ensuring efficiency requirements are met. Authors define five exit points on AlexNet and shows that their system is ale to meet strict delay constraints (200-400 ms) for inference of CIFAR-10 images, exploiting one edge device (Raspberry Pi 3) and on edge server (desktop PC).

The placement of early exit points needs to be carefully determined based on the available devices. To address this, Chiang et al. [44] extends BranchyNet by introducing an algorithm that dynamically identifies the optimal branch. The authors propose a dynamic programming solution to this NP-complete problem, allowing for efficient and effective placement of the early exit points.

The combination of split computing and early exit has also been addressed by Ebrahimi et al. [45]. The study focused on two simulated environments: a mobile topology comprising a SmartWatch, SmartPhone, BaseStation, and Cloud Server, and an IoT topology consisting of a Raspberry Pi, Micro Datacenter, and Cloud Servers. The primary contribution of their work lies in the proposal of a performance model capable of estimating both inference latency

and accuracy while considering various partitioning strategies and Early Exit placements. This model enables the adaptation of each device within the topology to achieve an optimal tradeoff between latency and accuracy.

An additional aspect that should be taken into account, considering the inherent instability of edge networks, is the potential indirect redundancy introduced by split computing and early exit techniques, which can prove beneficial in mobile edge networks. In the event of an internet connection loss, for instance, the edge device can still provide local outputs, albeit with reduced accuracy. This issue is addressed by Ju et al. in [46]. The authors proposed a system that can save a significant portion, up to 100%, of the affected frames during handovers by leveraging advanced frame saving schemes, frame choosing schemes, and frame repartition schemes. This approach is particularly suitable in mitigating the impact of frame loss in scenarios such as Vehicular Networks and applications involving drones, where handovers can lead to frame loss.

This theme is also considered in [47]. Authors proposed a novel algorithm called Scheduling Early Exit (SEE), which utilizes dynamic programming techniques to determine the optimal schedule with various early exit choices in the event of a service outage. This algorithmic approach allows for efficient decision-making regarding early exit strategies, ensuring effective DNN inference even when faced with service outage.

2.3.4 Distributed Computing

While previous research has primarily concentrated on hierarchical systems involving edge servers and cloud servers, it is essential to recognize that this is not the sole solution. A recent trend has emerged, focusing on edge device networks that do not incorporate servers.

Consequently, we explore the concept of distributing different segments of the employed model across multiple devices. This approach aims to leverage the unique capabilities of each device, enabling them to perform computations on a subset of the DNN layers. As a result, it effectively reduces the storage requirements for the model and minimizes the number of floating-point operations (FLOPs) required per input.

One of the first works proposing this approach is MoDNN [48], which presents a solution for accelerating DNN computations by partitioning pre-trained models onto multiple mobile devices, thereby reducing device-level computing costs and memory usage. Experimental results demonstrate the effectiveness of MoDNN: as the number of worker nodes increases from 2 (typical edge device - edge server topology) to 4, the system achieves a DNN computation speedup of 2.17x - 4.28x. This acceleration is not solely attributed to parallel execution but also to the reduction in data delivery time. The same key idea is behind Deepthings [49], which aims at partitioning the deep learning model and distributing the computation across multiple edge devices within a cluster. This allows the workload to be shared, enabling parallel and efficient inference. The paper introduces a dynamic load balancing mechanism that adapts the partitioning strategy based on the computational capabilities and availability of resources in the edge cluster.

Although there is limited existing literature on the investigation of distributed computing with partial offloading applied to various topologies comprising edge devices with different capabilities, other studies have focused on harnessing the diverse computational and storage capacities by utilizing different models for the same task, stored across multiple edge devices

within a network. A noteworthy example is presented by Si Salem et al. [50]: the authors proposed an inference network consisting of computing nodes, each capable of hosting specific pre-trained ML models. While we consider distributed computing to be a more efficient solution, the suggested topology served as inspiration for my research.

As anticipated in Section 2.3.3, [40] [41] [42] [45] have proposed the combined usage of early exit and split computing. However, all of these systems are based on hierarchical design with a pre-defined and static order of execution across devices (i.e. edge device $->$ edge server $->$ cloud server). This work aims at filling this gap, generalizing to systems which can also be only composed of edge devices, which cooperate in a dynamic way exploiting early exit and distributed computing.

CHAPTER 3

METHODOLOGY

3.1 Design details

As previously mentioned, the objective of this work is to create a system that can utilize the computation and communication abilities of a diverse range of devices with varying topologies. The system aims to enable distributed inference, taking advantage of early exit to minimize inference delay and reduce the overall system load. This is particularly important as the system is expected to operate with limited computational capabilities.

Our algorithms are specifically crafted to address two prevalent constraints in this field; (i) the accuracy constraint, which aims to achieve specific accuracy levels, and (ii) the input arrival rate constraint, which involves adapting the inference process to find the optimal accuracy tradeoff that satisfies the given rate of input arrivals.

Once the results for these two scenarios were gathered, we deploy systems that handle incoming input based on a Poisson process. These systems possess predetermined lengths for both input and output buffers. In case the output buffer reaches its maximum capacity, outputs are transferred back to the input queue to allow for local processing to continue. Similarly, if the input queue becomes full, new inputs are discarded. To address fluctuating input rates and prevent input dropping, the system employs a dynamic early exit policy, which is elaborated

in the following section. This policy adjusts the number of cases where early exit is performed based on the current input rate.

3.1.1 Early Exit policy

In order to provide flexibility, the early exit policy does not depend on the architecture and the number of early exit points. Early exit is achieved by introducing a fully connected layer after a specific layer i to ensure that the output has the appropriate dimensionality (corresponding to the number of classes). The decision to exit or continue the inference is based on a policy that compares the confidence level of each particular exit to a threshold value, denoted as **T**.

To calculate the **confidence** of each classification, the softmax function (as defined in Equation 2.2) is applied to the classification vector, which has a dimensionality equivalent to the number of classes in the dataset being used. The resulting value represents the probability of the input belonging to the most probable class.

If the confidence exceeds the threshold value (**T**), the inference process is halted, and the classification result is obtained. On the other hand, if the confidence falls below the threshold, the activation vector obtained from layer i serves as input to the subsequent layers of the model, allowing the inference to continue.

Focusing on threshold T, we implemented two approaches, corresponding to the two constraints previously introduced. The first system involves a fixed threshold value T and, subsequently, a fixed accuracy, while the rate of the input images change. On the other hand, the second one maintains a fixed rate of input images while varying the threshold value (**T**),

resulting in varying levels of accuracy. These systems will be thoroughly explored and analyzed in the upcoming sections.

3.2 Algorithms

The implementation of the codes in this work utilizes multithreaded programming. In this section, tables are provided to illustrate the assignment of threads to different scenarios on each device, along with the corresponding algorithms and functions employed. To comprehend these tables, it is necessary to introduce several variables utilized in the algorithms:

- threshold T represents the confidence threshold utilized the early exit policy, as described in section 3.1.1;
- *interval* refers to the inter-arrival rate at which input images are sequentially added to the input queue;
- *stats* comprise statistics computed at regular intervals (typically every 1 second) to obtain information about the state of each device. These statistics encompass the length of the input queue, RAM usage, and the computation time $compTime_i$ (for device i) for multiplying two 50x50 matrices. This matrix multiplication is performed to gather insights into the machine's state and inference speed;
- $commTime_i$ is the moving average over 5 seconds of the time required to complement the exchange of a 50x50 matrix with node i.

In addition, there are several hyperparameters that can be configured:

- a, b, and c represent coefficients that determine how the threshold T and the interval are adjusted based on the system state. These coefficients provide various options for adapting the reactivity speed. Excessively high values can result in system deadlock, while overly small values can lead to slower adaptation. In algorithms 2, 3, 6, and 7, we can customize these coefficients to achieve a faster or slower adjustment of the threshold and inter-arrival interval. When the queue is empty or minimally loaded, we decrease the inter-arrival interval or increase T by a factor of a. If it is slightly loaded, we employ coefficient b, which is expected to be smaller than a. Conversely, when the input queue is heavily loaded, we decrease T or increase the interval using coefficient c. In this scenario, we select a single coefficient that enables rapid adaptation to prevent system deadlock, as having excessive elements in the queue could lead to undesirable outcomes, which we strive to avoid whenever feasible.

- $minInterval$ denotes the minimum interval desired in the best-case scenario, when the device is not heavily loaded. Setting this parameter helps avoid excessively small intervals that could overload the system too quickly. In general, it should be declared as a value greater than zero to prevent negative inter-arrival times, which could potentially lead to exceptions when running the algorithms.

- $tQueue1$, $tQueue2$, and $tQueue3$ are thresholds for the load in the input and output (only in the distributed scenario) queues, which influence the adjustment of T and the interval. These thresholds can be correlated with application requirements (e.g., a maximum buffering delay) or device hardware constraints (e.g., memory). The thresholds are

directly linked to coefficients a, b, and c. They are employed to ascertain when the algorithm should increase or decrease T and the inter-arrival interval, as well as the extent to which these adjustments are made.

- *s1* and *s2* determine the duration of the waiting periods during the execution of algorithms 2, 3, 6, and 7. *s2* represents the time interval between each queue check, designed to avoid unnecessary checks when the queue load changes gradually rather than instantaneously. On the other hand, *s1* represents the waiting time after an adaptation is made, allowing the system sufficient time to react and consider the system's behavior after the new value is set, thereby preventing overly rapid modifications without proper consideration.

3.2.1 Local inference

Although the local inference is not the aim of this work, we will provide the algorithm we developed to have a fair comparison to the algorithms used for distributed inference.

Table I outlines the tasks required for local inference. Each task is intended to be executed independently and concurrently with the others. It includes the algorithms we designed, which will be explained in this section.

TABLE I: LOCAL INFERENCE TASKS.

Task 1	Manages the inference function (algorithm 1)
Task 2	Computes and shows `stats`
Task 3	Manages the input queue inserting elements according to the defined `interval`
Task 4	Modifies `interval` (algorithm 2) or `T` (algorithm 3)

Algorithms 1 manages the inference function: inference is performed on inputs stored in the input queue sequentially, eventually applying EE. Algorithms 2 and 3 manage the adaptation of the threshold and interval in the two scenarios to mitigate the possibility of reaching deadlock state of the device caused by its overloading.

In the case of a fixed confidence threshold T (Algorithm 2), the inter-arrival interval is decreased if the device is under a light load. This reduction in interval occurs in two distinct situations, which are determined by two low load levels represented by thresholds $tQueue1$ and $tQueue2$. On the other hand, if the device is overloaded (indicated by the input queue exceeding the threshold $tQueue3$), the interval is increased. This adjustment helps alleviate the device's load arising from incoming inputs. The same approach is exploited when considering fixed interval and a variable confidence threshold T (Algorithm 3): lower T corresponds to more early exit and consequently less computation.

The specific values for these thresholds ($tQueue1$, $tQueue2$, and $tQueue3$), can be chosen based on the capabilities of the individual device, particularly with respect to memory constraints.

Algorithm 1: Inference function in local scenario

```
while Input data is not the end flag do
    Compute the classification
    Store the used early exit point
    Check if classification is correct
end
```

Algorithm 2: inter-arrival interval adaptation

```
while Classification is running do
    if length(input queue) < tQueue1 & interval > minInterval then
        interval = interval - a*interval
        sleep s1 seconds
    end
    else if length(input queue) ≥ tQueue1 & length(input queue) < tQueue2 & interval
     > minInterval then
        interval = interval - b*interval
        sleep s1 seconds
    end
    else if length(input queue) > tQueue3 then
        interval = interval + c*interval
        sleep s1 seconds
    end
    sleep s2 seconds
end
```

Algorithm 3: confidence threshold adaptation

```
while Classification is running do
    if length(input queue) < tQueue1 then
        T = min(1, T + a*T)
        sleep s1 seconds
    end
    else if length(input queue) ≥ tQueue1 & length(input queue) < tQueue2 then
        T = min(1, T + b*T)
        sleep s1 seconds
    end
    else if length(input queue) > tQueue3 then
        T = max(0.5, T - c*T)
        sleep s1 seconds
    end
    sleep s2 seconds
end
```

3.2.2 Distributed inference

Regarding distributed inference, additional tasks need to be managed due to communication and the distribution of inference. To handle this, we decided to establish separate connections for each function and each device in the system. For example, there is a connection dedicated to exchanging activation vectors with device A, another for exchanging activation vectors with device B, and additional connections for exchanging statistics with device A, and so on. This approach ensures that different communication queues are in place, preventing interference between them. The algorithms 2 and 3 remain unchanged, except that the output queue is now included when assessing device load (algorithms 6 and 7).

The codes executed on the different devices used in the experiments are nearly identical, except for the presence of the original input data, which is exclusively owned by one device.

The general approach is as follows: when the device is under a light load and capable of handling the classification, we prioritize local execution. This is observed when using a smaller confidence threshold (while maintaining a fixed accuracy constraint) or having a small input rate (while keeping a fixed input rate), thus more classifications can be performed locally, as shown in Algorithm 4: if the input queue is empty, the output is re-put in the input queue, which means to continue the inference locally. However, if the device starts to become overloaded, we shift our focus to leveraging other devices. Load balancing becomes a crucial tradeoff in this process. Our objective is to respect the constraints while achieving optimal performance, while also avoiding excessive load on the receivers. This is important to prevent deadlock situations for both the receivers and the transmission channel.

In the next part of this section, we will illustrate our algorithms, highlighting the key aspects that contribute to achieving our goals.

Table Table II provides a comprehensive overview of the tasks essential for distributed inference. Each task is expected to be executed independently and simultaneously with the others.

Algorithm 4 is responsible for overseeing the inference process. It operates by sequentially analyzing each element present in the input queue. If the early exit mechanism is not applied, the intermediate result is placed back into the input queue, allowing the inference process to continue. However, if the input queue is not empty and the output queue has space, the intermediate result is placed in the output queue, where it remains ready to be transmitted to another device within the network.

TABLE II: DISTRIBUTED INFERENCE TASKS.

Task 1	Manages the inference function (algorithm 4)
Task 2	Manage the communications to send activation vectors (algorithm 5)
Task 3	Manage the communications to receive classification outputs
Tasks 4	Manage the communications to receive `stats`
Task 5	Computes $t_i, \forall i$ (eq. Equation 3.1)
Task 6	Manages the input queue inserting elements according to the defined `interval`
Tasks 7	Manage the communications to receive activation vectors
Tasks 8	Compute and update $commTime_i, \forall i$
Tasks 9	Modifies `interval` (algorithm 2) or `T` (algorithm 3)

Algorithm 5 is utilized to manage the transmission of activation vectors from the output queue (populated by Algorithm 4) to other devices within the network. Several factors come into play when determining the appropriate recipient device. Firstly, the load of the receiver device is checked by examining the input queue of that device (data contained in the received `stats`). If the input queue surpasses a predefined threshold, the device is not considered as a potential receiver to avoid overloading it.

The selection of device j is performed with a probability of 1 only if its corresponding t_j (computed according to equation Equation 3.1) is the smallest among all t_i values or if the output queue is highly loaded ($output\ queue > outputLoadThreshold$), thus we aim at exploiting as many receivers as possible. The significance of this criterion varies based on its computation: it indicates that the device has a low load (small input queue), efficient communication with the device, and low computational demand. This implies that the device is not heavily loaded

Algorithm 4: Inference function in distributed scenario.

```
while Input data is not the end flag do
    Compute the part of inference choosing the proper subnetwork[1]
    if confidence > T then
        Check if classification is correct
    end
    else
        if Input queue is empty or Output queue ≥ tOutQueue then
            Add the intermediate result to the input queue
        end
        else
            Add the intermediate result to the output queue
        end
    end
end
```

[1]The selection of the specific subnetwork (i.e. the specific layers to be used for inference) is determined based on the information associated with the input vector. This vector is flagged to indicate it is the original input with flag value of 0, 1 represents the output after EE point 1, and so on.

and, in a heterogeneous scenario, possesses favorable computational capabilities compared to other devices in the network.

Continuing with the aforementioned considerations, there are cases where device j is chosen as a receiver even if its t_j value is not the smallest. This selection is made with a probability that is proportional to the ratio between the minimum t_i and t_j. The purpose of this approach is to utilize all devices in the network and achieve a balanced distribution of the load. This is particularly important in situations where t_i values are similar to each other. By avoiding a stringent policy that would exclude all devices except one, we ensure a more equitable distribution of tasks across the network.

Algorithm 5: Activation vectors sending management.

(**Communication to node** j)
while *Input data is not the end flag* **do**
  if $InputQueue_j < tQueueJ$ **then**
    if $t_j < t_i, \forall i$ *or output queue* $>$ *outputLoadThreshold* **then**
      Send an activation vector from the output queue to device t_j
    end
    else
      Send an activation vector from the output queue to device j with probability $\frac{\min(t_i, \forall i)}{t_j}$
    end
  end
end

$$t_i = compTime_i * length(inputQueuei) + commTime_i \tag{3.1}$$

Algorithms 6 and 7 outline the adaptation of the threshold and interval in the two scenarios: fixed inter-arrival rate and fixed accuracy constraint. As already introduced, the primary objective is to prevent overloading of the device and mitigate potential deadlock situations.

The exploited mechanisms are the same described in 3.2.1 for the local inference, with the only difference that in this case the output queue dimension is also considering for threshold and interval adaptations.

As in the local inference case, *tQueue*1, *tQueue*2, and *tQueue*3, are chosen according to the capabilities of the used device, in particular considering its memory.

Algorithm 6: Inter-arrival interval adaptation

```
while Classification is running do
    if length(input queue) + length(output queue) < tQueue1 & interval > minInterval
     then
        interval = interval - a*interval
        sleep s1 seconds
    end
    else if length(input queue) + length(output queue) ≥ tQueue1 & length(input
     queue) + length(output queue) < tQueue2 & interval > minInterval then
        interval = interval - b*interval
        sleep s1 seconds
    end
    else if length(input queue) + length(output queue) > tQueue3 then
        interval = interval + c*interval
        sleep s1 seconds
    end
    sleep s2 seconds
end
```

Algorithm 7: confidence threshold adaptation

```
while Classification is running do
    if length(input queue) + length(output queue) < then
        T = min(1, T + a*T)
        sleep s1 seconds
    end
    else if length(input queue) + length(output queue) ≥ tQueue1 & length(input
     queue) + length(output queue) < tQueue2 then
        T = min(1, T + b*T)
        sleep s1 seconds
    end
    else if length(input queue) + length(output queue) > tQueue3 then
        T = max(0.5, T - c*T)
        sleep s1 seconds
    end
    sleep s2 seconds
end
```

3.2.3 Distributed inference and Poisson arrival process

After the design and testing of the aforementioned scenarios, we proceeded to evaluate our system using a Poisson arrival rate. In this case, the early exit policy and load balancing mechanisms remain the same as explained in section 3.2.2 for the fixed arrival rate scenario. However, there are some notable differences.

Firstly, we impose a limit on the size of the input queue, called *InputQueueSize*, and of the output queue. If the input queue becomes full, any new incoming inputs are dropped, resulting in a decrease in accuracy. On the other hand, the output queues are not subject to dropping. Instead, when an output queue reaches its limit, the inference process continues on

the device without moving intermediate results to the output queue. This approach ensures that the devices do not reach deadlock states as they are never overloaded.

Nevertheless, this adaptation introduces a tradeoff. As the number of dropped packets increases due to the arrival rate, there is a proportional decrease in accuracy. However, the devices avoids reaching deadlock situations and maintains performance by adjusting the confidence threshold T.

CHAPTER 4

EXPERIMENTAL SETUP AND IMPLEMENTATION

This chapter focuses on the architecture-specific implementation that we have designed for our study. We provide a detailed description of the models used and how we have adapted them to suit our requirements. Additionally, we offer insights into the topologies we have designed and tested. Furthermore, we elaborate on the implementation of the training process. Finally, we provide a brief overview of the physical setup, including the specific machines utilized for our experiments.

Once the system design was completed, we proceeded to implement it on actual devices that communicate via wireless connection. We replicated scenarios involving three or five devices with equivalent storage and computational capabilities, alongside an additional device featuring an edge server. This setup allowed us to compare the utilization of what we can call Peer Offloading (Partial Offloading to similar devices) with the more conventional hierarchical system Offloading. To conduct our experiments, we utilized two well-established models, MobileNetv2 and ResNet50, adapting them to our specific scenario and subsequently retraining the modified architectures.

4.1 Models

In this study, we chose to leverage two widely utilized lightweight models (introduced in Sec. 2.1.1) in edge computing with the goal of enhancing their performance using novel techniques

not previously discussed in their original publications [22] [23]. We will begin by describing these models in their original implementations, and subsequently, we will introduce the enhancements that we have incorporated.

4.1.1 MobileNetv2

Figure 5 gives a visual representation of the original architecture and the one implemented in this study.

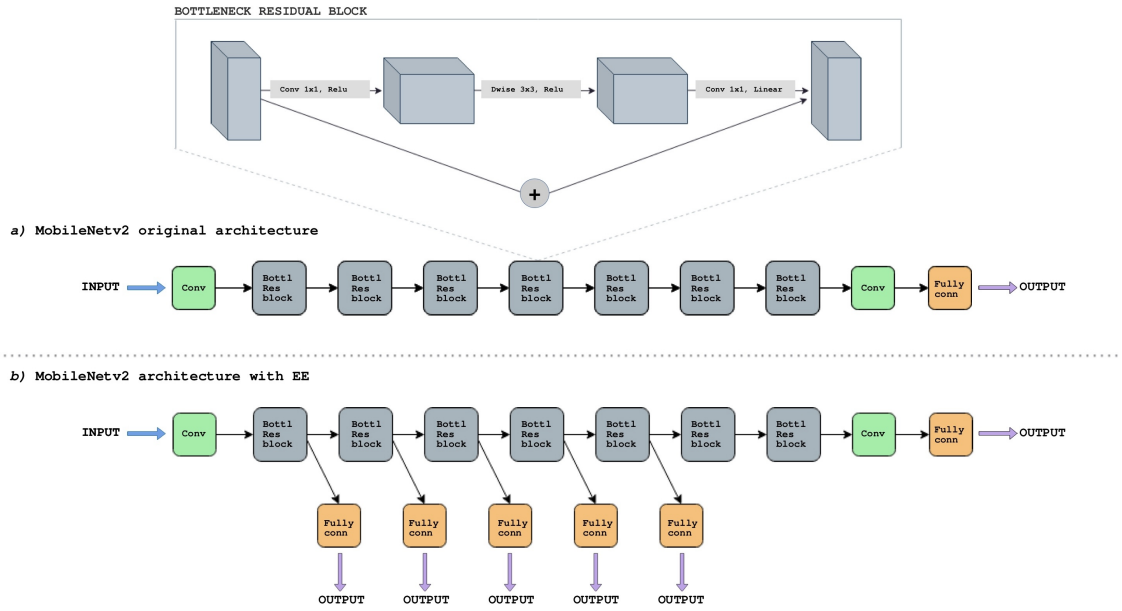

Figure 5: MobileNetv2: original architecture and architecture with EE.

4.1.1.1 Original implementation

The main idea behind MobileNetV2 is the use of a novel building block called the "inverted residual with linear bottleneck" (or Inverted Residual Block). This block consists of three main components: a lightweight depthwise separable convolution, an expansion layer, and a projection layer. Let's explore each of these components in more detail:

- Depthwise Separable Convolution: the depthwise separable convolution is a factorized convolution technique that splits the standard convolution into two separate operations: depthwise convolution and pointwise convolution. In the depthwise convolution, each input channel is convolved with a single convolutional filter, while in the pointwise convolution, a 1x1 convolution is used to combine the output channels obtained from the depthwise convolution. This separation significantly reduces the computational cost by decreasing the number of parameters and operations.
- Expansion Layer: the expansion layer is responsible for increasing the number of channels before applying the depthwise separable convolution. It uses a 1x1 convolutional layer with a nonlinear activation function (typically ReLU) to increase the dimensionality of the feature maps. This expansion helps capture more complex patterns and allows for better information flow through the network.
- Projection Layer: the projection layer is applied after the depthwise separable convolution to reduce the number of channels back to the desired value. It uses another 1x1 convolutional layer to compress the feature maps while maintaining the necessary representational capacity.

4.1.1.2 Adaptation to this study

As previously mentioned, this study focuses on two main elements: early exit and distributed computing. Regarding distributed computing, no modifications are required in the model architecture for it to function correctly. It suffices to divide the model into different submodels that can be stored separately on distinct devices.

In contrast, implementing early exit necessitates modifications in the network architecture. Specifically, to incorporate early exit points, the output of the hidden layers must be adjusted to the desired dimensionality (e.g., the number of classes in classification tasks), and the appropriate activation function must be applied to generate the desired output.

In this study, an early exit point is added after each Inverted Residual Block, except for the last two blocks, which are always computed together without an early exit point between them. This decision is based on the fact that only a few inputs result in the utilization of the full model. Consequently, it was deemed more important to segment the model at the earlier layers.

The choice of adding early exit points after entire Bottleneck Residual Blocks, without splitting them, is driven by the nature of these blocks. They terminate in a sum operation with the input of the block itself, as depicted in Figure 5.

4.1.2 ResNet

After evaluating ResNet50, ResNet101, and ResNet152, I concluded that ResNet50 provided satisfactory accuracy for the given dataset, prompting my decision to choose it.

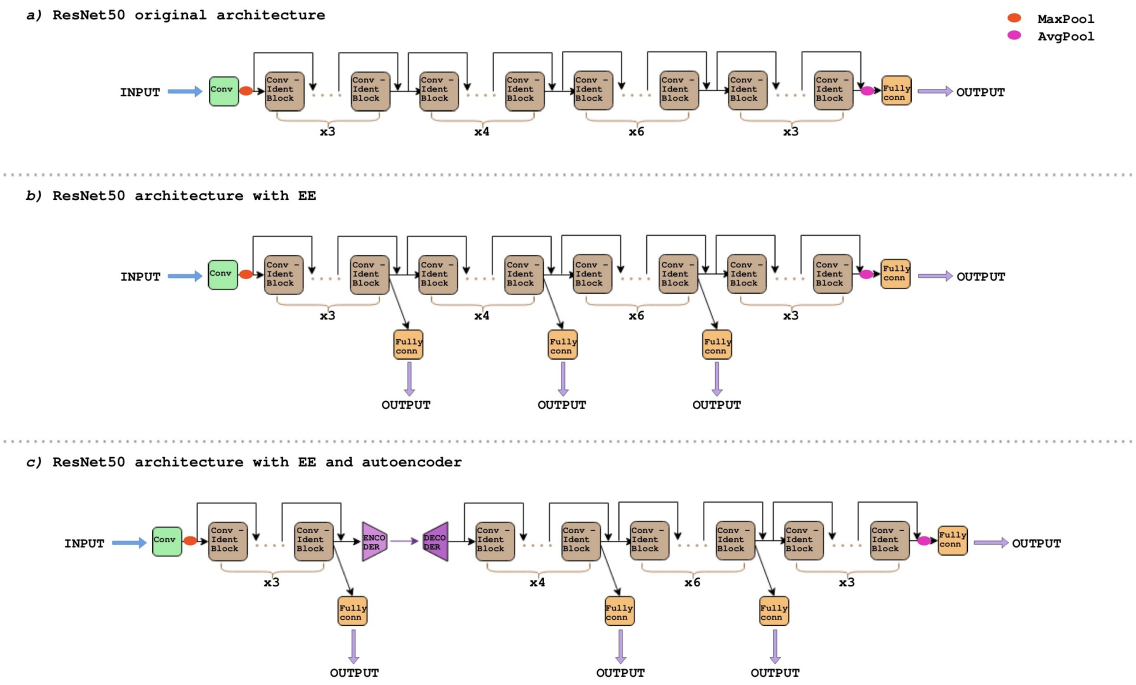

Figure 6: ResNet50: original architecture, architecture with EE and autoencoder.

Figure 6 provides a graphical depiction of both the original architecture and the architecture utilized in this study.

4.1.2.1 Original implementation

ResNet50 consists of 50 layers, hence its name. These layers can be broadly categorized into several types:

- Convolutional Layers: the network starts with a convolutional layer that performs filtering operations on the input image to extract low-level features. It is followed by several

blocks, each containing multiple convolutional layers. These layers progressively extract more complex and abstract features from the input data.

- Bottleneck Blocks: the core building blocks of ResNet50 are the bottleneck blocks. Each block consists of three convolutional layers: a 1x1 projection layer to reduce the number of input channels, a 3x3 convolutional layer for feature extraction, and another 1x1 projection layer to restore the number of channels. These blocks help in reducing the computational complexity of the network while maintaining expressive power.
- Identity Blocks: in addition to the bottleneck blocks, ResNet50 also includes identity blocks. An identity block is similar to the bottleneck block but does not have the dimension reduction step. It consists of three convolutional layers with the same number of input and output channels. Identity blocks allow the network to learn identity mappings, facilitating the training of deeper architectures.
- Pooling Layers: after the convolutional layers, ResNet50 employs average pooling to reduce the spatial dimensions of the feature maps. This downsampling operation helps in reducing the computational burden and capturing more global information.
- Fully Connected Layers: towards the end of the network, there are fully connected layers responsible for performing classification. They take the flattened feature maps and map them to the desired number of output classes.

4.1.2.2 Adaptation to this study

The main aspects of distributed computing and early exit are similar to what is explained in 4.1.1.2 and 3.1.1, referring to MobileNetv2, including the early exit policy based on confidence and threshold T.

In the case of MobileNetv2, the architecture incorporates Shortcut connections, which dictate the placement of early exit points. After careful consideration, we opted to incorporate three early exit points, with each one positioned after a set of identical Bottleneck Blocks. Additionally, the decision to deviate from a regular placement of early exit points, considering the required FLOPs, was driven by the observation that the initial layers are always computed, while the subsequent layers become less relevant due to the preceding early exit points. Hence, it seemed advantageous to offload the first subnetwork (layers before the first early exit point).

The positioning of early exit points must take into account both the computational reduction and accuracy of the network that incorporates them, requiring a tradeoff to be found. Specifically, for this architecture, it could be suggested to introduce an additional early exit point after the first convolutional layer, prior to the building blocks. Although we considered this option, the results obtained over 10,000 samples led us to dismiss this solution. It should be noted that, as depicted in Figure 7, a high level of confidence does not always correspond to high accuracy. In fact, the proposed early early exit point (placed after the first convolutional layer) resulted in an increased number of early exit points for each confidence threshold but led to significantly lower accuracies (10-20% less). Despite the potential reduction in inference

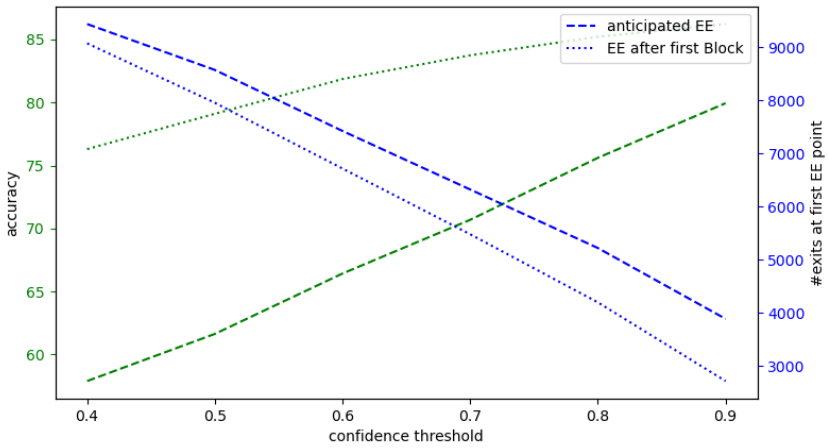

Figure 7: Accuracy and exits number with anticipated early exit and early exit after first Block vs confidence thresholds.

delay offered by this early early exit point, we decided against its usage in order to maintain higher accuracy levels after careful deliberation.

After conducting initial experiments in a distributed environment, it became evident that the activation vector following the first early exit point was excessively large. As a result, the message exchange between two devices, occurring after the first early exit point, emerged as a performance bottleneck, significantly limiting the system's overall performance. Consequently, we made the decision to devise a lightweight autoencoder to diminish the dimensionality of the messages, facilitating faster data exchange. The details of the autoencoder design can be found in Figure 8. It is worth noting that the number of required Floating Point Operations (FLOPs) differs between encoding and decoding. The encoding process entails approximately

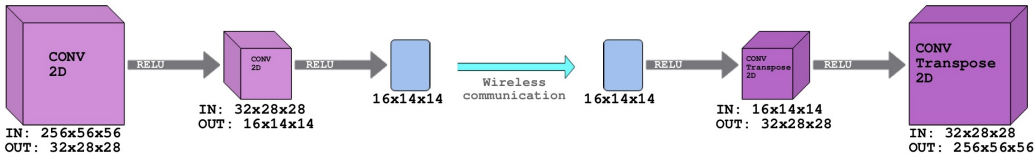

Figure 8: Autoencoder used with ResNet.

59 million FLOPs out of the total 807 million FLOPs of the first subnetwork (7.3%), whereas decoding requires 218 million FLOPs. This asymmetry in computation allocation aligns with the reasoning behind minimizing computations in the first subnetwork, which is more frequently utilized.

In contrast to the conventional approach with autoencoders, we did not train these layers independently by minimizing a similarity loss function. Instead, we trained them collectively as part of the entire network. The main objective was not to achieve a precise reproduction of the activation vector after encoding and decoding. Rather, the focus was on obtaining an activation vector that could be correctly classified as the original one, thus they do have to be as close as possible.

Utilizing the autoencoder architecture, we achieved a substantial reduction in dimensionality for the activation vector after the initial subnetwork, shrinking it from (256, 56, 56) to (16, 14, 14). Consequently, the overall message dimensionality experienced a remarkable decrease from 3.2MB to 13.3KB, enabling significant compression. In the worst-case scenario, where the first

(EE) is never utilized, and the full network is always exploited for inference, the employment of the autoencoder results in a an accuracy loss up to 2.5%, however this effected is mitigated by EE.

4.2 Topologies

As stated in section 2.3.3, based on our current understanding, there is a lack of existing literature on systems that combine early exit with distributed computing in flexible non-hierarchical topologies. This work aims to address this gap by proposing a system that extends the approach previously studied in the literature. The proposed system considers a network of heterogeneous devices without any path constraints for inference execution.

In scenarios where devices exhibit significant differences in capabilities, the system will naturally gravitate towards heavy utilization of the most capable device. This convergence results in a typical system configuration consisting of one edge device and one edge server.

Considering the benefits of distributed computing alone, it already offers advantages in terms of system load management by enabling parallelization of inference across different subnetworks within the same model.

To showcase the flexibility and adaptability of our algorithms, we created four distinct distributed environments in this study, as depicted in Figure 9. The environments consist of: (1) a setup with a single edge device and an edge server, (2) two setups involving three wireless-connected devices with different topologies, specifically a mesh topology and a unidirectional ring topology considering the activation vectors, and (3) a mesh topology setup with five devices. Each device incorporates subnetworks within their models, allowing them to process individual

subnetworks and determine whether to apply the early exit or not. If necessary, they can transmit activation vectors to another device or continue the inference process locally. The only distinction between these devices is that only one of them stores the input data and thus requires the first subnetwork. However, the system can be easily expanded to include more devices with input data. By sharing the first device's information (IP address and port), the classification output can be sent to it regardless of where the inference process concludes.

In our study, we leveraged the memory capacities of the devices employed, enabling us to store and utilize multiple subnetworks on each device. In a more restricted scenario, a simple approach to reduce the memory requirement would be to allocate different subnetworks to separate devices. However, this approach carries the inherent risk of an uneven distribution of the computational workload, potentially leading to a device becoming a bottleneck for the entire system.

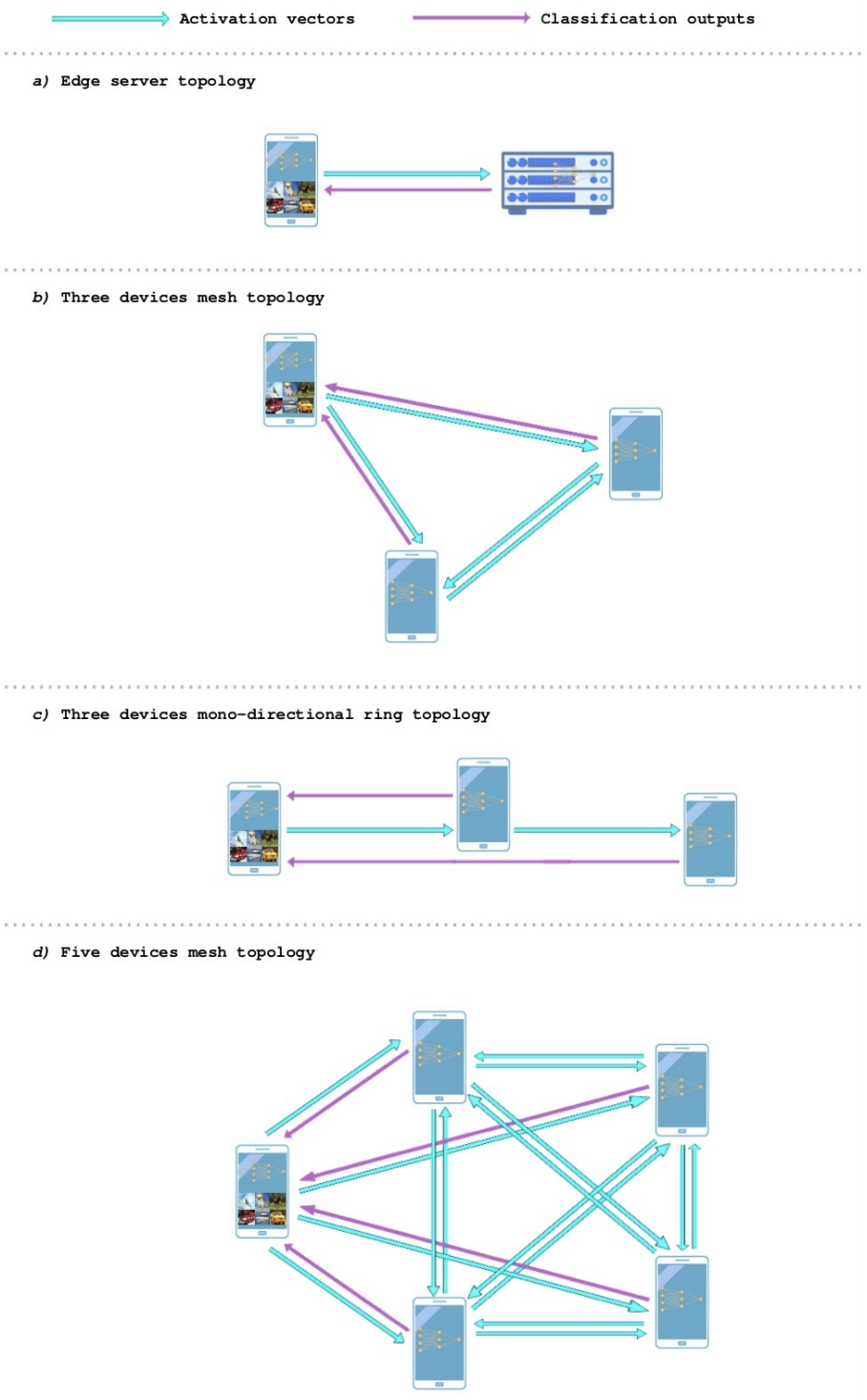

Figure 9: Distributed environments employed in this study.

4.3 Training

Training in the distributed environment was not conducted for this work, as the main focus was on inference. The training process was performed on a server, and the trained models were subsequently loaded onto the devices used in the experiments.

Moreover, it is worth noting that the primary objective of this work was not to optimize the training process itself. While it is possible to improve the accuracies of the models through various training techniques, it is important to emphasize that the obtained results should be interpreted in terms of a comparison between local and distributed moments, with or without the use of EE. The focus is on evaluating the performance and efficiency of the distributed inference framework, rather than achieving the highest absolute accuracy.

In our model training, we employed the widely used Cross Entropy Loss function (Equation 4.1), which is commonly utilized for classification tasks. This loss function quantifies the disparity between the predicted class ($\hat{y_i}$) probabilities and the actual class labels (y_i): it increases as the predicted probability diverges from the actual label.

$$\mathit{CrossEntropyLoss}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{4.1}$$

Considering the presence of multiple classification points, including the outputs from early exit points and the full model, we employed a cumulative loss approach. This method entailed aggregating the losses from all classifications, utilizing the loss formulation defined in Equation Equation 4.2. Within this equation, the coefficient c_e was introduced to weigh the loss at

each early exit point and the full network, enabling flexibility in assigning varying levels of importance to different classification points. In our case, we set all c_e equal to 1.

$$Loss(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{e=1}^{E} c_e(-\sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))) \tag{4.2}$$

4.4 Physical setup and codes implementation

To conduct our experiments, we utilized the NVIDIA® Jetson Nano™, a commonly used platform for AI development on edge devices. The Jetson Nano™ is equipped with the NVIDIA Maxwell architecture, featuring 128 NVIDIA CUDA® cores, a Quad-core ARM Cortex-A57 MPCore processor, and 4 GB of 64-bit LPDDR4 memory operating at 1600MHz with a bandwidth of 25.6 GB/s. For our edge server, we employed a GS65 Stealth 9SE laptop. This laptop is equipped with an Intel® i9-9880H processor and an NVIDIA® GeForce RTX™ 2060 graphics card with 6GB GDDR6 memory. It also boasts 16 GB of DDR4-SDRAM memory. These hardware configurations were chosen to support our experimental setup and ensure sufficient computational power for the tasks at hand.

Regarding device communication, we utilized wireless connectivity for our experiments. Specifically, we leveraged the UIC Wifi network to facilitate communication between the devices involved in our setup.

The implementation of the ideas and algorithms presented in this work was done entirely in Python. PyTorch was employed for implementing the Deep Neural Network (DNN) architectures. Socket and Pickle were used for communication purposes, facilitating the translation

of data into and from bit representation. Additionally, multi-thread programming was accomplished using the Threading module.

CHAPTER 5

RESULTS AND ANALYSIS

[1] In this chapter, we present the outcomes achieved by our system when utilizing CIFAR-10, utilizing a dataset that comprises 10,000 test images. Each test has been performed five times, and we report the average result along with its standard deviation. We employed the models described in Section 4.1, namely MobileNetv2 and ResNet50, along with the configurations specified in Section 4.2. Specifically, we refer to Figure 9 for the network topologies. Initially, we evaluated our system on a single device with a fixed inter-arrival interval and fixed confidence threshold T (consequently fixed accuracy). The obtained inter-arrival time when using fixed confidence threshold is proportional to the inference time required for each image, and these values are actually very similar. We also assessed it on the first three topologies, one which consists of a device-edge server and two topologies composed of three devices. Subsequently, we examined our system using a Poisson arrival rate and a limited input queue size, considering all the presented topologies, including the mesh topology with five devices.

5.1 Fixed inter-arrival interval and fixed confidence threshold T

In this section, we will present the outcomes achieved for the initial two scenarios discussed in Chapter 3.

---

[1] All the results depicted in the plots of this chapter are also provided as tables in the appendix.

Table IV contains the chosen values for the hyperparameters and coefficients present in the algorithms introduced in section 3.2, which are set to manage the load balancing and the threshold T and interval adaptation. As a conservative choice, at the beginning of each test with varying confidence threshold the value of T was set as 0.5 and the inter-arrival interval was set as 50 ms when using fixed T.

TABLE III: HYPERPARAMETERS AND COEFFICIENTS USED IN TESTS.

tQueue1	10
tQueue2	30
tQueue3	60
tOutQueue	50
tQueueJ	50
a	0.2
b	0.1
c	0.2

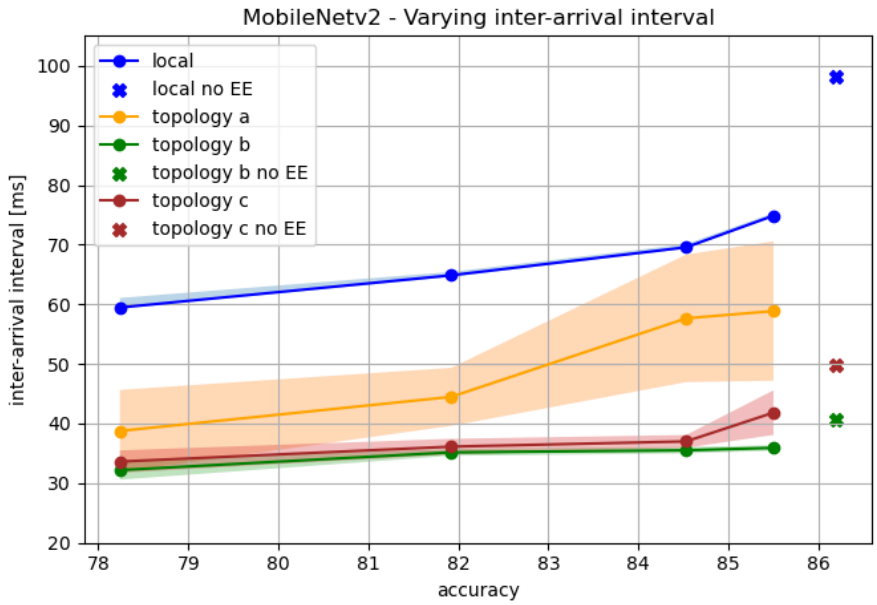

Figure 10: Results for MobileNetv2 with fixed confidence threshold T and varying inter-arrival interval.

5.1.1 MobileNetv2

Figure 10 presents the outcomes achieved by employing various fixed confidence thresholds (0.6, 0.7, 0.8, and 0.9) and for the model without early exit (referred to as 'no EE'), using MobileNetv2. As predicted, it is evident that the inference performed without early exit achieves greater accuracy but requires a longer inference time. As expected, employing lower confidence thresholds (resulting in decreased accuracies) enables the system to handle a larger input flow (smaller inter-arrival interval) with reduced inference time. Specifically, when examining the two topologies consisting of three devices (b and c), it is observed that employing a confidence

threshold of 0.7, for instance (which results in approximately 4.3% accuracy loss compared to inferences obtained with the full model), allows the system to achieve an inter-arrival interval 13.4% and 27.45% lower than when employing the full model. Even when considering a threshold of 0.9, the advantage remains significant, with the interval being 11.5% and 16% lower while experiencing only a 0.7% accuracy loss.

An intriguing observation is that topology a and topology c exhibit higher standard deviation values. In our view, this can be attributed to the fact that, with the first device having only one receiver device, the system is more susceptible to network variability and the influence of input order. This makes it easier to overload the device, resulting in performance degradation. Conversely, due to the conservative communication algorithm we implemented (where both receivers are utilized for activation vectors if the output queue is sufficiently large), topology b is less affected by differences in input sequence, resulting in a smaller standard deviation.

Regarding the system with a fixed and constant inter-arrival interval and varying threshold, the corresponding results are depicted in Figure 11. It should be noted that the intervals' values differ between different topologies. This deliberate variation aims to provide a clearer understanding of the intervals wherein we observed rapid changes in behavior. As for the minimal inter-arrival interval, the smallest value employed for each topology represents the minimum limit (with granularity of 5 ms) we could utilize while avoiding deadlock. For example, a 30 ms inter-arrival interval in topology a leads to deadlock due to excessive memory usage in the queues. The same holds true for a 35 ms inter-arrival interval in topology c and so forth. It is noteworthy that in topology a, the system can handle smaller inter-arrival intervals compared

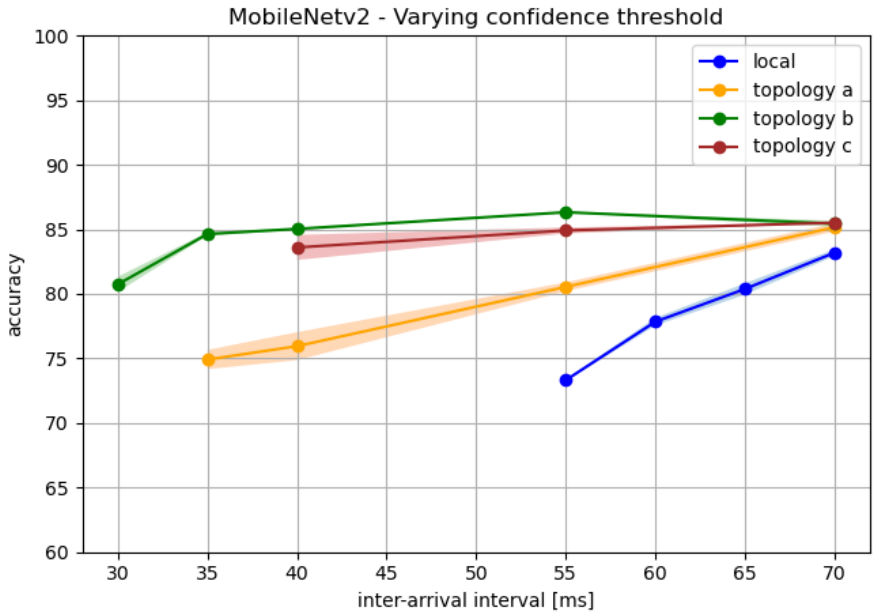

Figure 11: Results using MobileNetv2 with fixed inter-arrival interval and varying threshold T.

to topology c. This is related to the fact that, in the case of topology c, the second device of the topology (which has lower capabilities than the edge server) is the one causing deadlock. On the other hand, topology b can handle smaller inter-arrival intervals than topology a, as in the latter case, it is the first device that experiences deadlock in the output queue management due to having only one receiver. While it is possible to propose the use of different policies and hyperparameters for each topology to optimize their performance individually, we opted for using the same policies to ensure a fairer comparison between them.

Comparing the performance of different topologies with fixed confidence thresholds, we observe that topology b exhibits superior performance, while topologies a and c display higher standard deviations. Our algorithms enable the systems to outperform local inference, having better inference delays and consequently higher accuracies. As an example, when considering a shared inter-arrival interval of 55 ms, the distributed environment accuracies are significantly higher, with topology a, b, and c achieving improvements of 7.2%, 11.3%, and 11.6%, respectively.

5.1.2 ResNet50

When analyzing the results obtained from ResNet50, the first observation to note is that the accuracy differs between local and distributed scenarios when using the same confidence threshold. This is due to a decrease in accuracy in the distributed scenario caused by the utilization of the autoencoder.

Let's examine Figure 12, which displays the results obtained using a fixed confidence threshold. Similarly to the previous case, we observe a higher standard deviation for topologies a and b. This is particularly evident in topology a, which consists of only two devices, even though one of them is an edge server. In general, the algorithms adapt well to all topologies, with the distributed environment outperforming local inference. For instance, when comparing similar accuracy levels, such as using T=0.7 for local inference and T=0.8 for distributed environments, we observe significant improvements in terms of inter-arrival interval (and thus inference delay). Specifically, topologies a, b, and c require inter-arrival times that are 41.6%, 54%, and 20.4% shorter, respectively, while maintaining slightly higher accuracy (+0.32%).

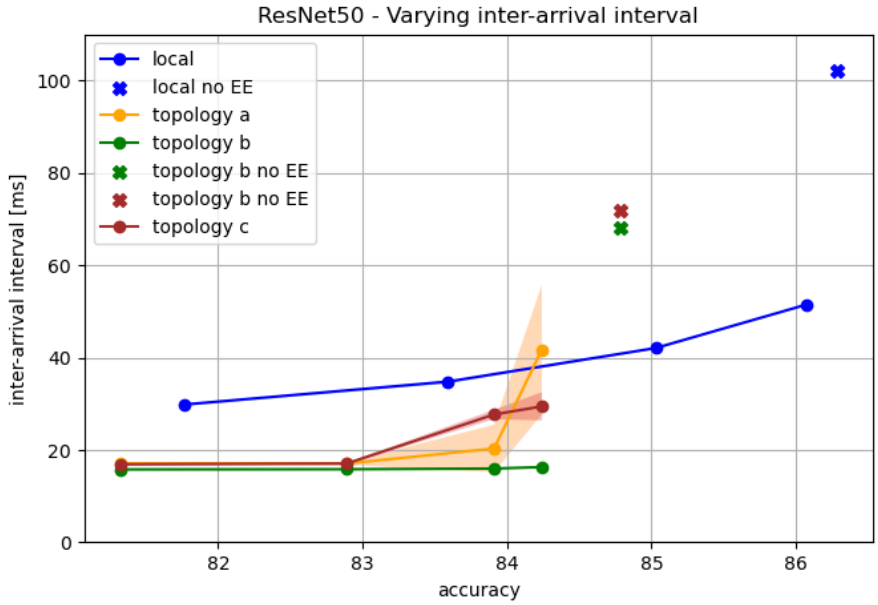

Figure 12: Results using ResNet50 with fixed confidence threshold T and varying inter-arrival interval.

The usage of early exit proves to be highly effective. For example, considering the highest tested confidence threshold (T=0.9), topologies b and c experience interval reductions of 76% and 59%, respectively, compared to the inference performed using the full model. This comes at the cost of only a 0.54% accuracy loss. When comparing the two extreme cases of local inference without early exit and topology b using early exit, we observe a reduction of 84% in the inter-arrival interval (with T =0.9) and an acceptable accuracy drop of 2.04%.

The results obtained when utilizing the full model versus early exit inferences are even better than those obtained with MobileNetv2. This is related to the fact that, with ResNet50, the

number of early exits is higher, reaching, for instance, (using $T = 0.6$, $T = 0.7$, $T = 0.8$ and $T = 0.9$) 67%, 55%, 42%, and 27% inference performed at the first early exit point, compared to 24%, 13%, 7%, and 2% in the previous tested model.

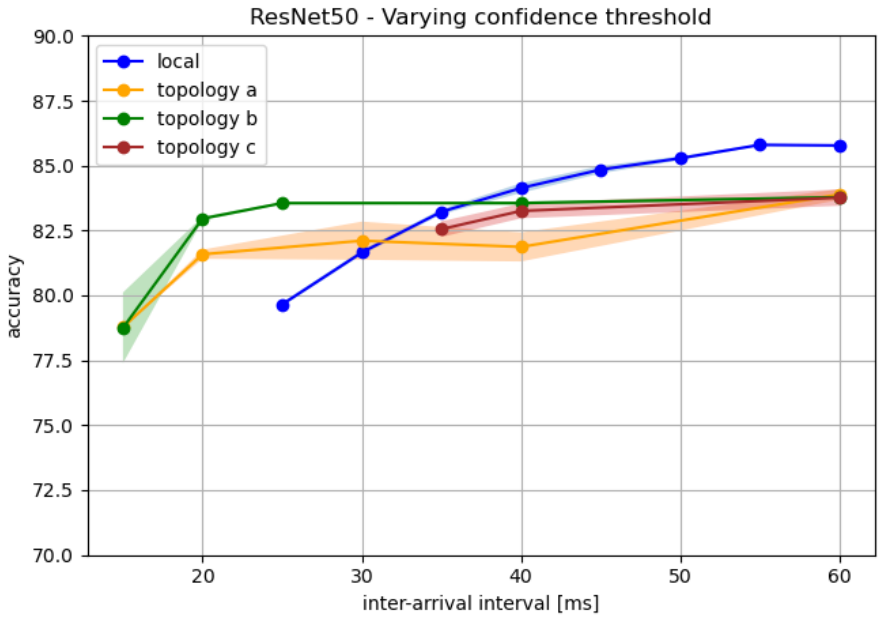

Figure 13: Results using ResNet50 with fixed inter-arrival interval and varying threshold T.

The results presented in Figure 13 demonstrate the outcomes achieved by utilizing ResNet50 with varying confidence thresholds. Starting from the highest levels of accuracy, it is observed that local inference, as previously discussed, yields superior accuracy due to the absence of

the autoencoder required in a distributed environment. Consequently, for applications without strict delay constraints (i.e., inter-arrival intervals larger than 40ms), this solution proves to be the most optimal.

Nevertheless, when considering shorter inter-arrival intervals, better performance is observed in the distributed environment. Specifically, the algorithms adapt well to all topologies, resulting in improved performance for topology b, which can handle smaller intervals without reaching a deadlock state as quickly as topology c. Interestingly, topology c reaches a deadlock state at a larger interval compared to local inference, primarily because the second device is the one causing the deadlock and due to the computational overhead related to autoencoder usage and communication. This occurrence is absent in topology a, as the edge server possesses greater resources than the second and third devices in topology c.

Topology a and c exhibit similar trends in their results. However, topology b surpasses topology a due to superior management of the output queue, leveraging two receivers instead of one. Consequently, it can be deduced that for topology a, the bottleneck is represented by the first device, which only becomes the bottleneck for topology b at smaller intervals. This can be observed when analyzing the accuracy achieved by the system at a 15 ms interval, where the results obtained using the edge server or topology b are nearly identical in terms of accuracy.

5.2 Poisson arrival rate

In this section, we will showcase the outcomes achieved while employing a Poisson arrival rate for the input. Both the output queue and the input queue have a capacity of 60. It is important to note that when the output queue becomes overloaded, intermediate vectors are not dropped. Instead, if the output queue reaches its maximum capacity, the inference process continues locally without adding intermediate outputs to the output queue for transmission. Conversely, if the input queue becomes full, new inputs are discarded, resulting in a decrease in accuracy. All the following experiments are performed starting with empty input queue.

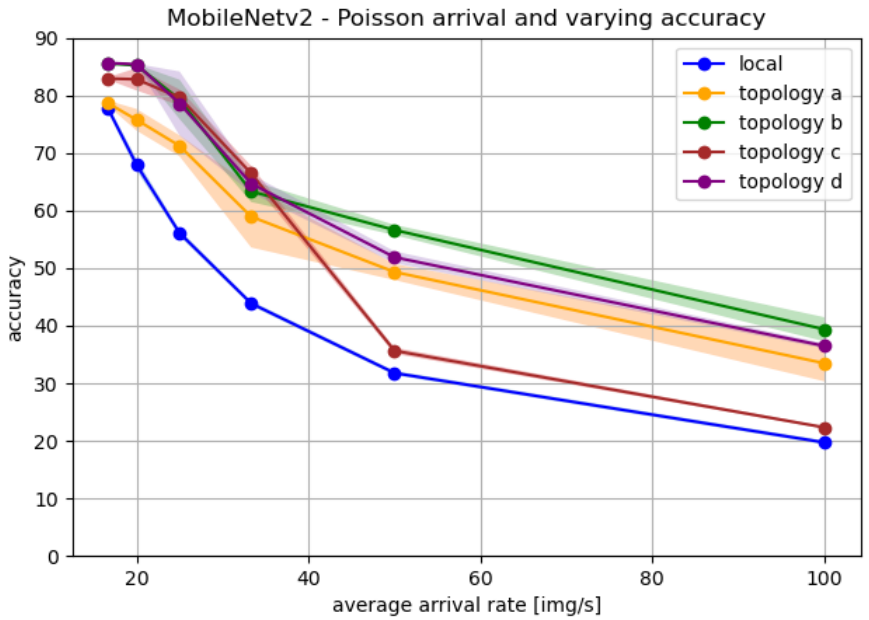

Figure 14: Results for ResNet50 with Poisson arrival and varying threshold T.

The results presented in Figure 14 are obtained by employing an input flow characterized by inter-arrival times following a Poisson distribution. The average inter-arrival times (μ) are set to 10ms, 20ms, 30ms, 40ms, 50ms, and 60ms for MobileNetv2, which correspond to image rates of 16.7, 20, 25, 33.3, 50, and 100 images per second, respectively. In addition, topology d, a mesh topology consisting of 5 edge devices, was also tested in this scenario.

As expected, our algorithms demonstrate that all distributed environments outperform local inference in every case. An interesting observation is related to the edge server system, which achieves a lower maximum accuracy. This outcome can be attributed to the less efficient communication in this particular topology, resulting in a higher number of dropped packets. Apart from this observation, the systems perform similarly under low arrival rates. However, as the arrival rate increases, significant differences become apparent. For instance, topology b achieves an accuracy that is three times higher than local inference when the rate is 100 images per second. Even at $\mu = 50$ (20 images per second), the distributed environments exhibit accuracies that are 7.8%, 14.9%, 17.3%, and 17.5% higher than the local case.

Similarly to the previous cases analyzed, the more constrained topology c yields poorer performance compared to the others, particularly when the arrival rate exceeds 40 ms. This can be attributed to the limited resources of the second device, which affects the overall system performance.

Contrary to our expectations, utilizing a larger number of devices (topology d) does not yield better results. This is due to the fact that, with dedicated connections, the communication overhead becomes increasingly significant, leading to suboptimal resource utilization,

especially for the first device, which becomes overloaded. On the other hand, topology b seems to exploit the resources more effectively. These observations lead us to propose alternative to the communication protocol employed, in section 6.1.

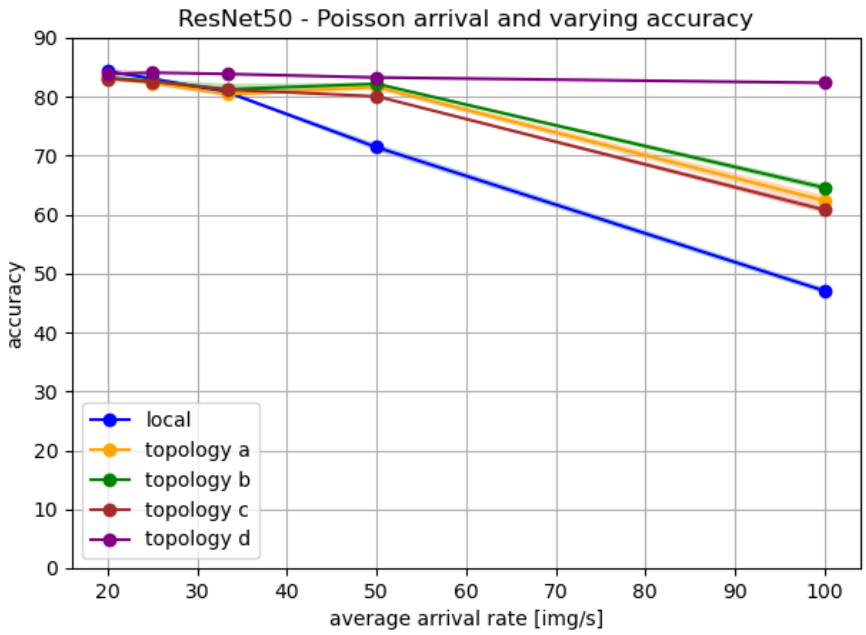

Figure 15: Results using MobileNetv2 with Poisson arrival and varying threshold T.

The results depicted in Figure 15 showcase the outcomes obtained by employing ResNet50 and a Poisson arrival rate with μ values of 10 ms, 20 ms, 30 ms, 40 ms, and 50 ms (corresponding to input rates of 100, 50, 33.3, 25, and 20 images per second).

In this scenario, we observe a consistent behavior across all topologies, aligning with our expectations. Our algorithms enable each system to achieve higher accuracies compared to local inference. For example, with an input rate of 50 images per second, the distributed environments attain accuracies that are 10.2%, 10.8%, 9.6%, and 11.8% higher, while with 100 images per second, they demonstrate accuracies that are 15.3%, 17.5%, 13.7%, and 35% higher.

Moreover, in this particular case, topology d outperforms all the others. This can be attributed to the utilization of the autoencoder, which facilitates a significant reduction in activation vector dimensions and enables topology d to effectively manage the output queue without becoming overloaded, as observed when using MobileNetv2. Notably, at the highest input rate (100 images per second), topology d achieves an accuracy that is 21.4% higher than the second-best topology (topology b).

CHAPTER 6

FUTURE WORK AND CONCLUSION

6.1 Future work

We have successfully demonstrated the effective combination of distributed inference and early exit across various topologies, even those that do not involve a dedicated server or hierarchical systems. However, we have observed instances where an increase in the number of devices involved can lead to a performance decrease, contrary to our expectations. We have attributed this issue to the resources utilized by each device to handle communication and data exchange with other devices in the network.

To address this, it would be valuable to introduce a logic within our system that dynamically determines, in the context of a large mesh topology, whether all connections need to be utilized or selectively maintained or discarded. This logic could be implemented in an online manner and enhanced through the adoption of lightweight communication protocols, such as MQTT commonly used in IoT. For instance, we could analyze the tradeoff obtained as the latency for acquiring statistics and data from other devices increases (in contrast to our current approach prioritizing low communication latency through Socket communication), while simultaneously reducing the number of connections required. This would enable our system to scale effectively to larger topologies while demanding fewer resources on each individual device.

We are confident that these modifications, even with the inclusion of new communication protocols, would not hinder the proper functioning and adaptability of our algorithms to diverse topologies. Our algorithms are designed to be directly independent of specific communication protocols and mediums, allowing for seamless integration with the proposed modifications.

Furthermore, another enhancement that would increase the flexibility of our system is the implementation of an automated method to determine early exit points. In existing literature, many tests are conducted using fixed early exit points, such as at 15%, 30%, 45%, 60%, 75%, and 90% of the network. However, as demonstrated in 4.1.2.2, incorrect positioning of early exits can result in significant decreases in accuracy. This is particularly true because high confidence does not always correspond to high accuracy. Therefore, we believe that implementing a logic that intelligently selects the optimal early exit points for any given model, without specific predefined percentages, could improve accuracy outcomes.

6.2 Conclusion

We introduced a novel system that integrates early exit and distributed learning techniques to enable faster and collaborative inference in edge networks. Our objective was to propose a comprehensive and flexible approach that is not limited to hierarchical networks like most existing proposals in the literature. Instead, we aimed to develop a solution that can adapt to diverse and heterogeneous topologies, accommodating devices with varying capabilities. In order to accomplish this, we introduced generalized algorithms that are independent of specific topologies, devices, or implementations. These algorithms only necessitate the configuration of only a few hyperparameters that are tailored to the specific scenario.

The noteworthy aspect is that both the early exit policy and load balancing mechanisms are dynamically adjusted in real-time. This adaptation is facilitated by the exchange of data among all devices in the topology, considering the state and load of the network as well as communication requirements, although this data exchange incurs a minimal overhead compared to the data exchange necessary for the inference task. In particular, the early exit policy relies on a confidence threshold, denoted as T, which determines whether the early exit should be applied or not. This threshold is dynamically adjusted during runtime on each device, with different devices potentially having different values of T, depending on their specific load. The decision to continue the inference locally or to transmit intermediate activation vectors to another device in the network for distributed computing is based on the collective knowledge of all devices present in the network that can receive activation vectors from the sender.

In the tested topologies, we observed that increasing the number of devices does not always enhance the capability of the systems, especially when dealing with lightweight models like MobileNetv2, which are already optimized for resource-constrained devices. However, when handling larger models such as ResNet, noticeable improvements are evident, such as when using a topology with five devices compared to one with only three devices. Nevertheless, it is important to note that the devices we utilized are specifically designed for AI at the edge and possess relatively good performance capabilities. It would be intriguing to evaluate our system on larger topologies comprising more resource-constrained devices. In this scenario, when dealing with a larger number of connections, particularly those associated with exchanging statistics for each device, we believe that utilizing communication protocols like MQTT would be a favorable option. Although there would be a tradeoff in terms of increased latency, it would enable the system to scale effectively to accommodate a substantial number of receivers.

Furthermore, as we aimed not to achieve an exceptionally high absolute accuracy value, we did not extensively explore modifying the training process of the models. Specifically, we proposed a cost function with coefficients that could be utilized to prioritize certain early exit points over others, potentially encouraging the system to perform early exits in specific points more frequently than others. Consequently, it would be interesting to experiment with tailored solutions that emphasize certain early exit points during training and observe the resulting impact on the overall accuracy.

Our algorithms proved to be effective in all the tested topologies and with both models utilized. Notably, significant improvements were observed, such as average inference times for

ResNet being approximately 6 times smaller when utilizing only 3 edge devices compared to a non-distributed experiment without early exit on the same edge devices, with only a 1.8% drop in accuracy. Additionally, a three-device system using MobileNetv2 was able to handle 2.2 times more images per second compared to a single device, while maintaining the same level of accuracy.

In conclusion, we are confident that the concept presented in our research provides a foundation for exploring new approaches in adapting to flexible environments and diverse topologies. This concept holds potential for practical applications in various scenarios, such as military operations or situations involving service outages characterized by low communication rates and isolated local area networks (LANs) without access to cloud services. The proposed system has the potential to significantly impact real-world applications by improving performance in resource-constrained environments. It specifically addresses scalability and generalizability concerns across different models and topologies, achieving notable advancements in inference time while maintaining a tradeoff with accuracy.

APPENDICES

RESULTS FOR MOBILENETV2

TABLE B.I: MOBILENETV2 AND FIXED CONFIDENCE THRESHOLD T.

		T = 0.6	**T = 0.7**	**T = 0.8**	**T = 0.9**	**no EE**
One device	Accuracy	78.24	81.92	84.53	85.50	86.20
	Average interval [ms]	59.44	64.85	69.54	74.84	98.21
	Interval stdev	0.14	0.26	0.19	0.12	0.22
Topology a	Accuracy	78.24	81.92	84.53	85.50	
	Average interval [ms]	38.73	44.45	57.62	58.85	
	Interval stdev	6.88	4.87	10.69	11.69	
Topology b	Accuracy	78.24	81.92	84.53	85.50	86.20
	Average interval [ms]	32.17	35.14	35.49	35.90	40.57
	Interval stdev	1.60	0.57	0.53	0.46	1.41
Topology c	Accuracy	78.24	81.92	84.53	85.50	86.20
	Average interval [ms]	33.60	36.10	36.98	41.81	49.76
	Interval stdev	1.88	1.31	1.08	3.73	7.97

TABLE B.II: MOBILENETV2 AND VARYING CONFIDENCE THRESHOLD T.

Interval [ms]		**30**	**35**	**40**	**55**	**60**	**65**	**70**
One device	Average accuracy				73.32	77.84	80.38	83.18
	Accuracy stdev				0.065	0.30	0.44	0.26
Topology a	Average accuracy		74.91	75.95	80.54			85.15
	Accuracy stdev		0.74	1.08	0.34			0.32
Topology b	Average accuracy	80.74	84.64	85.03	86.32			85.43
	Accuracy stdev	0.58	0.13	0.074	0.045			0.23
Topology c	Average accuracy			83.59	84.91			85.51
	Accuracy stdev			0.97	0.23			0.030

TABLE B.III: MOBILENETV2 AND POISSON ARRIVAL PROCESS.

Average interval [ms]		**60**	**50**	**40**	**30**	**20**	**10**
One device	Average accuracy	77.72	67.94	56.08	43.86	31.77	19.71
	Accuracy stdev	0.56	0.98	0.57	0.42	0.38	0.35
Topology a	Average accuracy	78.64	75.75	71.22	58.92	49.32	33.46
	Accuracy stdev	0.45	1.88	1.82	5.35	1.41	3.13
Topology b	Average accuracy	85.54	85.21	79.30	63.31	56.61	39.36
	Accuracy stdev	0.09	0.12	3.41	1.83	0.89	2.10
Topology c	Average accuracy	82.93	82.79	79.63	66.43	35.64	22.32
	Accuracy stdev	0.21	1.95	1.38	1.30	0.53	0.22
Topology d	Average accuracy	85.63	85.39	78.58	64.66	51.88	36.48
	Accuracy stdev	0.14	0.08	5.56	1.59	1.02	0.46

.

RESULTS FOR RESNET50

TABLE B.IV: RESNET50 AND FIXED CONFIDENCE THRESHOLD T.

		T = 0.6	**T = 0.7**	**T = 0.8**	**T = 0.9**	**no EE**
One device	Accuracy	81.77	83.59	85.03	86.07	86.28
	Average interval [ms]	29.86	34.80	42.08	51.45	101.99
	Interval stdev	0.35	0.44	0.25	0.19	0.67
Topology a	Accuracy	81.32	82.89	83.91	84.24	
	Average interval [ms]	17.12	17.06	20.33	41.72	
	Interval stdev	0.14	0.30	5.13	13.99	
Topology b	Accuracy	81.32	82.89	83.91	84.24	84.78
	Average interval [ms]	15.80	15.83	15.98	16.30	67.99
	Interval stdev	0.13	0.13	0.06	0.14	0.89
Topology c	Accuracy	81.32	82.89	83.91	84.24	84.78
	Average interval [ms]	16.90	17.07	27.69	29.44	71.76
	Interval stdev	0.26	0.16	1.02	3.06	0.75

TABLE B.V: RESNET50 AND VARYING CONFIDENCE THRESHOLD T.

Interval [ms]		**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**	**55**	**60**
One device	Average accuracy			79.65	81.65	83.22	84.14	84.85	85.28	85.80	85.78
	Accuracy stdev			0.08	0.09	0.10	0.19	0.15	0.08	0.04	0.03
Topology a	Average accuracy	78.77	81.58		82.11		82.87				83.89
	Accuracy stdev	0	0.18		0.74		0.56				0.20
Topology b	Average accuracy	78.75	82.96	83.55			83.56				83.78
	Accuracy stdev	1.36	0.07	0.44			0.60				0.48
Topology c	Average accuracy					82.55	83.25				83.77
	Accuracy stdev					0.30	0.29				0.32

TABLE B.VI: RESNET50 AND POISSON ARRIVAL PROCESS.

Average interval [ms]		**50**	**40**	**30**	**20**	**10**
One device	Average accuracy	84.292	83.03	80.74	71.41	47.06
	Accuracy stdev	0.20	0.28	0.25	0.64	0.57
Topology a	Average accuracy	83.09	82.29	80.40	81.62	62.36
	Accuracy stdev	0.20	0.28	0.25	0.64	0.57
Topology b	Average accuracy	83.20	82.64	81.24	82.18	64.55
	Accuracy stdev	0.10	0.35	0.95	0.19	0.60
Topology c	Average accuracy	82.99	82.56	81.19	80.05	60.79
	Accuracy stdev	0.09	0.10	0.13	0.12	0.58
Topology d	Average accuracy	83.83	84.07	83.85	83.25	82.34
	Accuracy stdev	0.99	0.01	0.11	0.28	0.09

CITED LITERATURE

1. Zhang, X., Wang, Y., and Shi, W.: pCAMP: Performance Comparison of Machine Learning Packages on the Edges.

2. Ali, M., Anjum, A., Yaseen, M. U., Zamani, A. R., Balouek-Thomert, D., Rana, O., and Parashar, M.: Edge Enhanced Deep Learning System for Large-Scale Video Stream Analytics. In 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC), pages 1–10, May 2018.

3. Kar, G., Jain, S., Gruteser, M., Bai, F., and Govindan, R.: Real-time traffic estimation at vehicular edge nodes. In Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC '17, pages 1–13, New York, NY, USA, October 2017. Association for Computing Machinery.

4. `https://engineering.fb.com/2016/11/08/android/delivering-real-time-ai-in-the-palm-of-your-hand/`.

5. Liu, C., Cao, Y., Luo, Y., Chen, G., Vokkarane, V., Yunsheng, M., Chen, S., and Hou, P.: A New Deep Learning-Based Food Recognition System for Dietary Assessment on An Edge Computing Service Infrastructure. IEEE Transactions on Services Computing, 11(2):249–261, March 2018. Conference Name: IEEE Transactions on Services Computing.

6. Lin, Z. Q., Chung, A. G., and Wong, A.: EdgeSpeechNets: Highly Efficient Deep Neural Networks for Speech Recognition on the Edge, November 2018. arXiv:1810.08559 [cs, eess, stat].

7. `https://gsacom.com/paper/economic-impact-emerging-passenger-economy-intel/`.

8. Chen, M., Tian, Y., Fortino, G., Zhang, J., and Humar, I.: Cognitive Internet of Vehicles. Computer Communications, 120:58–70, May 2018.

9. Liang, L., Ye, H., and Li, G. Y.: Toward Intelligent Vehicular Networks: A Machine Learning Framework. IEEE Internet of Things Journal, 6(1):124–135, February 2019. Conference Name: IEEE Internet of Things Journal.

CITED LITERATURE (continued)

10. Chang, W.-J., Chen, L.-B., and Su, K.-Y.: DeepCrash: A Deep Learning-Based Internet of Vehicles System for Head-On and Single-Vehicle Accident Detection With Emergency Notification. IEEE Access, 7:148163–148175, 2019. Conference Name: IEEE Access.

11. Hsu, C. C.-H., Wang, M. Y.-C., Shen, H. C., Chiang, R. H.-C., and Wen, C. H.: FallCare+: An IoT surveillance system for fall detection. In 2017 International Conference on Applied System Innovation (ICASI), pages 921–922, May 2017.

12. Miraftabzadeh, S. A., Rad, P., Choo, K.-K. R., and Jamshidi, M.: A Privacy-Aware Architecture at the Edge for Autonomous Real-Time Identity Reidentification in Crowds. IEEE Internet of Things Journal, 5(4):2936–2946, August 2018. Conference Name: IEEE Internet of Things Journal.

13. Liu, L., Zhang, X., Qiao, M., and Shi, W.: SafeShareRide: Edge-Based Attack Detection in Ridesharing Services. In 2018 IEEE/ACM Symposium on Edge Computing (SEC), pages 17–29, Seattle, WA, USA, October 2018. IEEE.

14. Kaya, Y., Hong, S., and Dumitras, T.: Shallow-Deep Networks: Understanding and Mitigating Network Overthinking, May 2019. arXiv:1810.07052 [cs, stat].

15. Teerapittayanon, S., McDanel, B., and Kung, H. T.: BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks, September 2017. arXiv:1709.01686 [cs].

16. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, April 2017. arXiv:1704.04861 [cs].

17. Zhang, X., Zhou, X., Lin, M., and Sun, J.: ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices, December 2017. arXiv:1707.01083 [cs].

18. Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, November 2016. arXiv:1602.07360 [cs].

19. Tan, M. and Le, Q. V.: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, September 2020. arXiv:1905.11946 [cs, stat].

20. Redmon, J., Divvala, S., Girshick, R., and Farhadi, A.: You Only Look Once: Unified, Real-Time Object Detection, May 2016. arXiv:1506.02640 [cs].

CITED LITERATURE (continued)

21. Lin, J., Chen, W.-M., Lin, Y., Cohn, J., Gan, C., and Han, S.: MCUNet: Tiny Deep Learning on IoT Devices, November 2020. arXiv:2007.10319 [cs].

22. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C.: MobileNetV2: Inverted Residuals and Linear Bottlenecks, March 2019. arXiv:1801.04381 [cs].

23. He, K., Zhang, X., Ren, S., and Sun, J.: Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385 [cs].

24. Krizhevsky, A., Sutskever, I., and Hinton, G. E.: ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6):84–90, May 2017.

25. Wang, C.-Y., Bochkovskiy, A., and Liao, H.-Y. M.: YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, July 2022. arXiv:2207.02696 [cs].

26. Görmez, A., Dasari, V. R., and Koyuncu, E.: E$ˆ2$CM: Early Exit via Class Means for Efficient Supervised and Unsupervised Learning. In 2022 International Joint Conference on Neural Networks (IJCNN), pages 1–8, July 2022. arXiv:2103.01148 [cs, stat].

27. Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

28. Zhou, W., Xu, C., Ge, T., McAuley, J., Xu, K., and Wei, F.: BERT Loses Patience: Fast and Robust Inference with Early Exit.

29. Görmez, A. and Koyuncu, E.: Pruning Early Exit Networks, July 2022. arXiv:2207.03644 [cs].

30. Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J.: Live Video Analytics at Scale with Approximation and Delay-Tolerance.

31. Pearson, K.: LIII. On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 2(11):559–572, November 1901. Publisher: Taylor & Francis.

32. Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A.: Extracting and composing robust features with denoising autoencoders. In Proceedings of the 25th international

CITED LITERATURE (continued)

conference on Machine learning - ICML '08, pages 1096–1103, Helsinki, Finland, 2008. ACM Press.

33. Sutskever, I., Vinyals, O., and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, December 2014. arXiv:1409.3215 [cs].

34. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y.: Generative Adversarial Networks, June 2014. arXiv:1406.2661 [cs, stat].

35. Wang, W., Huang, Y., Wang, Y., and Wang, L.: Generalized Autoencoder: A Neural Network Framework for Dimensionality Reduction. In 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, pages 496–503, June 2014. ISSN: 2160-7516.

36. Hinton, G. E. and Salakhutdinov, R. R.: Reducing the Dimensionality of Data with Neural Networks. Science, 313(5786):504–507, July 2006.

37. Masci, J., Meier, U., Cireşan, D., and Schmidhuber, J.: Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction. In Artificial Neural Networks and Machine Learning – ICANN 2011, eds. T. Honkela, W. Duch, M. Girolami, and S. Kaski, volume 6791, pages 52–59. Berlin, Heidelberg, Springer Berlin Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

38. Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P.: MAUI: making smartphones last longer with code offload. In Proceedings of the 8th international conference on Mobile systems, applications, and services, pages 49–62, San Francisco California USA, June 2010. ACM.

39. Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L.: Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, pages 615–629, New York, NY, USA, April 2017. Association for Computing Machinery.

40. Li, E., Zeng, L., Zhou, Z., and Chen, X.: Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing, October 2019. arXiv:1910.05316 [cs].

CITED LITERATURE (continued)

41. Teerapittayanon, S., McDanel, B., and Kung, H.: Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pages 328–339, June 2017. ISSN: 1063-6927.

42. Laskaridis, S., Venieris, S. I., Almeida, M., Leontiadis, I., and Lane, N. D.: SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, pages 1–15, September 2020. arXiv:2008.06402 [cs, stat].

43. Zeng, L., Li, E., Zhou, Z., and Chen, X.: Boomerang: On-Demand Cooperative Deep Neural Network Inference for Edge Intelligence on the Industrial Internet of Things. IEEE Network, 33(5):96–103, September 2019. Conference Name: IEEE Network.

44. Chiang, C.-H., Liu, P., Wang, D.-W., Hong, D.-Y., and Wu, J.-J.: Optimal Branch Location for Cost-effective Inference on Branchynet. In 2021 IEEE International Conference on Big Data (Big Data), pages 5071–5080, December 2021.

45. Ebrahimi, M., Veith, A. D. S., Gabel, M., and De Lara, E.: Combining DNN partitioning and early exit. In Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking, pages 25–30, Rennes France, April 2022. ACM.

46. Ju, W., Yuan, D., Bao, W., Ge, L., and Zhou, B. B.: eDeepSave: Saving DNN Inference using Early Exit During Handovers in Mobile Edge Environment. ACM Transactions on Sensor Networks, 17(3):1–28, August 2021.

47. Wang, Z., Bao, W., Yuan, D., Ge, L., Tran, N. H., and Zomaya, A. Y.: SEE: Scheduling Early Exit for Mobile DNN Inference during Service Outage. In Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pages 279–288, Miami Beach FL USA, November 2019. ACM.

48. Mao, J., Chen, X., Nixon, K. W., Krieger, C., and Chen, Y.: MoDNN: Local distributed mobile computing system for Deep Neural Network, 2017. Pages: 1396-1401.

49. Zhao, Z., Barijough, K. M., and Gerstlauer, A.: DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2348–2359, November 2018.

CITED LITERATURE (continued)

50. Salem, T. S., Castellano, G., Neglia, G., Pianese, F., and Araldo, A.: Towards Inference Delivery Networks: Distributing Machine Learning with Optimality Guarantees, December 2021. arXiv:2105.02510 [cs].

51. Chen, G., Parada, C., and Heigold, G.: Small-footprint keyword spotting using deep neural networks. In 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 4087–4091, May 2014. ISSN: 2379-190X.

52. Han, S., Mao, H., and Dally, W. J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, February 2016. arXiv:1510.00149 [cs].

53. He, Y., Zhang, X., and Sun, J.: Channel Pruning for Accelerating Very Deep Neural Networks, August 2017. arXiv:1707.06168 [cs].

54. Yang, T.-J., Chen, Y.-H., and Sze, V.: Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 6071–6079, Honolulu, HI, July 2017. IEEE.

55. Hinton, G., Vinyals, O., and Dean, J.: Distilling the Knowledge in a Neural Network, March 2015. arXiv:1503.02531 [cs, stat].

56. Mirzadeh, S.-I., Farajtabar, M., Li, A., Levine, N., Matsukawa, A., and Ghasemzadeh, H.: Improved Knowledge Distillation via Teacher Assistant, December 2019. arXiv:1902.03393 [cs, stat].

57. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D.: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2704–2713, Salt Lake City, UT, June 2018. IEEE.

58. Li, G., Liu, L., Wang, X., Dong, X., Zhao, P., and Feng, X.: Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge, December 2018. arXiv:1812.06426 [cs].

59. Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, November 1998. Conference Name: Proceedings of the IEEE.

CITED LITERATURE (continued)

60. Cheong Took, C. and Mandic, D.: Weight sharing for LMS algorithms: Convolutional neural networks inspired multichannel adaptive filtering. Digital Signal Processing, 127:103580, July 2022.

61. Prabhavalkar, R., Alsharif, O., Bruguier, A., and McGraw, I.: On the Compression of Recurrent Neural Networks with an Application to LVCSR acoustic modeling for Embedded Speech Recognition, May 2016. arXiv:1603.08042 [cs].

62. Chen, P., Si, S., Li, Y., Chelba, C., and Hsieh, C.-J.: GroupReduce: Block-Wise Low-Rank Approximation for Neural Language Model Shrinking.

63. Seifi, S., Jha, A., and Tuytelaars, T.: Glimpse-Attend-and-Explore: Self-Attention for Active Visual Exploration, August 2021. arXiv:2108.11717 [cs].

64. Khan, L. U., Yaqoob, I., Tran, N. H., Kazmi, S. M. A., Dang, T. N., and Hong, C. S.: Edge-Computing-Enabled Smart Cities: A Comprehensive Survey, October 2020. arXiv:1909.08747 [cs].

65. Zhang, P., Gan, P., Chang, L., Wen, W., Selvi, M., and Kibalya, G.: DPRL: Task Offloading Strategy Based on Differential Privacy and Reinforcement Learning in Edge Computing. IEEE Access, 10:54002–54011, 2022. Conference Name: IEEE Access.

66. Olakanmi, O. O. and Odeyemi, K. O.: Trust-aware and incentive-based offloading scheme for secure multi-party computation in Internet of Things. Internet of Things, 19:100527, August 2022.

67. Li, J., Zhang, Z., Yu, S., and Yuan, J.: Improved Secure Deep Neural Network Inference Offloading with Privacy-Preserving Scalar Product Evaluation for Edge Computing. Applied Sciences, 12(18):9010, January 2022. Number: 18 Publisher: Multidisciplinary Digital Publishing Institute.

68. Mao, Y., Hong, W., Wang, H., Li, Q., and Zhong, S.: Privacy-Preserving Computation Offloading for Parallel Deep Neural Networks Training. IEEE Transactions on Parallel and Distributed Systems, 32(7):1777–1788, July 2021. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

69. McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and Arcas, B. A. y.: Communication-Efficient Learning of Deep Networks from Decentralized Data, January 2023. arXiv:1602.05629 [cs].

CITED LITERATURE (continued)

70. Jeong, H.-J., Jeong, I., Lee, H.-J., and Moon, S.-M.: Computation Offloading for Machine Learning Web Apps in the Edge Server Environment. In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pages 1492–1499, July 2018. ISSN: 2575-8411.