

Cloud/Edge Inference for Deep Neural Networks

Marion Sbai

Balliol College

University of Oxford

A thesis presented for the degree of

Doctor of Philosophy

Friday 13th October, 2023

Acknowledgements

First and foremost, I would like to sincerely thank my supervisors Prof. Niki Trigoni and Dr. Andrew Markham for their continued advice and invaluable guidance throughout the project, which made me understand how to conduct a high-quality research. I would like to address them a special thanks for their human qualities, their patience and their support in difficult times. Thanks to their trust, I was able to complete my research work and overcome moments of doubt.

I am also grateful to all members of Cyber Physical System (CPS) research group, especially Risqi, who accompanied me in the first moments of my thesis and allowed me to improve my skills more quickly. I would also like to thank Professors Nicholas Lane, Ivan Martinovic, Ivan Flechais and Tam Vu who, through their comments at my Transfer of Status and Confirmation of Status, provided me with useful feedback to improve my thesis. Thanks also to Professor Samy Meftali, whose support was decisive.

Finally, I greatly thank my family, especially my parents, for their care and relentless support, and my brother, who always supported me during our time in Oxford.

Abstract

In the past decade, deep learning has become one of the most dominant data analytic approaches due to its capability to achieve impressively high accuracies on a variety of key computing tasks. Some of these tasks need intensive computation, especially when image and video data captured from cameras are input to the predictive model. In traditional systems like video surveillance, these tasks are offloaded to the cloud server, which automatically process the collected data by employing deep learning algorithms. This solution has limitations, most importantly high latency and network load, impacting the performance of real-time applications. The alternative solution, which consists in directly embedding deep neural networks (DNN) on the edge device (e.g. camera), is greatly limited by the amount of computation and storage resources. Since then, some research has examined approaches which distribute the inference of machine learning algorithms between the edge and the cloud. However, despite some promising results, several fundamental drawbacks still exist in terms of accuracy, efficiency, and applicability to resource-constrained environments.

The work presented in this thesis tackles these shortcomings by proposing a novel partitioning approach for DNN inference between the edge and the cloud. This is the first work to consider simultaneous optimization of both the memory usage at the edge and the size of the data to be transferred over the wireless link. The experiments were performed on two different network architectures, MobileNetV1 and VGG16. The proposed approach makes it possible to execute part of the network on very constrained devices (e.g., microcontrollers), and under poor network conditions (e.g., LoRa) whilst retaining reasonable accuracies.

The second contribution deals with DNN distribution on a cloud/multi-edge execution platform. By including early-exits and bottlenecks in the neural network, the solution

we propose makes it possible to distribute computation according to time and accuracy requirements. In scenarios where the communication with the cloud is very limited (e.g. LoRa), it allows a 80% improvement in average latency compared to a traditional cloud solution.

Finally, we deal with power-constrained devices. We propose an energy consumption estimation and optimization approach for DNNs with multiple early exits in a platform composed of several edge devices and a cloud server. The approach utilizes machine learning combined with dynamic linear programming techniques to estimate the energy consumption of different DNN configurations and identify the optimal configuration that minimizes energy consumption while maintaining the desired accuracy level.

Statement of Originality

This thesis is written in accordance with the regulations for the degree of Doctor of Philosophy. The thesis has been composed by myself and has not been submitted in any previous application for any degree. The work presented in this thesis is my own.

Table of Contents

1 Introduction **1**

1.1 Context and motivation 1

1.2 Research challenges 4

1.3 Research objectives 7

1.4 Contributions 7

1.5 Thesis structure 9

1.6 Publications 10

2 Background **11**

2.1 Introduction 11

2.2 Edge computing 13

2.2.1 Heterogeneity of edge devices 14

2.2.2 Specific DNN architectures for edge computing 16

2.2.3 DNN compression techniques 18

2.2.4 Discussion 19

2.3 Cloud/Single-edge DNN partitioning 20

2.3.1 Data amplification issue 21

2.3.2 Intermediate data compression 23

2.3.3 Neural network optimisation at the edge 26

2.3.4 Discussion 27

2.4 Dynamic input-dependent inference in deep neural networks . . . 28

2.4.1 Single execution platform 28

2.4.1.1 Dynamic depth 29

2.4.1.2 Dynamic width 35

2.4.1.3 Dynamic routing 36

2.4.2 Dual execution platform 38
2.4.3 Discussion 39
2.5 Cloud/Multi-edge DNN partitioning 40
2.6 Summary 42

3 Cloud/Single-edge DNN partitioning 44
3.1 Introduction 44
3.2 Architecture design 47
3.2.1 Inflection points 47
3.2.2 Architecture description 49
3.3 Learning methods 51
3.3.1 Training the student network at the edge 51
3.3.2 Auto-encoding intermediate data 52
3.3.3 Fine-tuning 52
3.3.4 Training process 53
3.4 Implementation details 54
3.5 Sensitivity analysis 55
3.5.1 The Impact of Cutting at Different Layers 55
3.5.2 Trade-off Between Communication Cost and Accuracy . . 57
3.5.3 Trade-off Between Memory Size and Accuracy 58
3.6 Comparison with Related Works 62
3.6.1 Encode (E) 62
3.6.2 Cut-Encode (Cut, Encode (CE)) 63
3.7 Summary 64

4 Cloud/Multi-edge image dependent DNN patitioning 66
4.1 Introduction 66
4.2 System overview 70

4.2.1 General description 70
4.2.2 Metrics 70
4.2.2.1 Accuracy 71
4.2.2.2 Time 72
4.3 Detailed design 72
4.3.1 Step 1 : DNN architecture deployment and training (Offline) 72
4.3.2 Step 2 : Workload sharing algorithm (offline) 74
4.3.2.1 Fixed-threshold strategy (FT) 74
4.3.2.2 Linear programming optimisation (LP) 77
4.3.3 Step 3: Distributed execution - Dynamic Threshold Configuration (Online) 80
4.4 Evaluation 81
4.4.1 Implementation details 81
4.4.2 Architecture deployment and training (Step 1) 82
4.4.3 Workload sharing algorithm (Step 2) 83
4.4.4 Distributed execution (Step 3) 84
4.4.4.1 Dynamic threshold configuration 84
4.4.4.2 When and how the LP and Dynamic Threshold should be used? 87
4.4.4.3 Time/Accuracy analysis 87
4.4.4.4 Convergence 89
4.5 Summary 90

5 Dynamic optimisation of energy consumption 91
5.1 Energy consumption estimate 92
5.1.1 Introduction 92
5.1.2 General principle 92
5.1.3 Energy consumption estimate during run time 93

5.1.3.1 Identification of the energy-consuming activities 93
5.1.3.2 Estimation of the energy consumption of each activity . . . 95
5.1.3.3 Energy consumption estimation flow . . . 96
5.2 Energy consumption optimisation . . . 98
5.2.1 Integer linear programming model . . . 98
5.2.1.1 Controlled parameters . . . 98
5.2.1.2 Non-controlled parameters . . . 98
5.2.1.3 LP model . . . 99
5.2.2 Design decisions coming out of the model . . . 100
5.2.3 Limitations of the linear program . . . 100
5.2.4 Dynamic linear programming . . . 101
5.2.4.1 Optimisation flow . . . 102
5.2.5 Dynamic threshold adaptation . . . 105
5.2.6 Optimisation algorithm execution . . . 105
5.3 Case study . . . 106
5.3.1 Activities counting . . . 106
5.3.1.1 Processor . . . 106
5.3.1.2 Memory . . . 107
5.3.1.3 Communication . . . 108
5.3.2 Architecture elements activities energy consumption estimation . . . 108
5.3.2.1 Processor . . . 109
5.3.2.2 Memory . . . 111
5.3.2.3 Communication . . . 112
5.3.3 Energy consumption optimisation flow . . . 112
5.3.3.1 Initialisation iteration . . . 112

5.3.3.2 Other iterations . . . 115
5.4 Results and discussion . . . 115
5.4.1 Sensitivity to the energy scenario . . . 115
5.4.2 Effectiveness of the dynamic threshold adaptation . . . 116
5.4.3 Sensitivity to the presence/absence of permanent energy sources . . . 118
5.4.4 Comparison with cloud-only processing . . . 119
5.4.5 Limitations . . . 120
5.5 Summary . . . 121

6 Conclusion and future work **122**
6.1 Conclusion . . . 122
6.2 Directions for future work . . . 124
6.2.1 Mobile cameras . . . 124
6.2.1.1 Research problem . . . 124
6.2.1.2 Ideas for potential solutions . . . 124
6.2.2 Fault-tolerant multi-camera system and task migration . . . 125
6.2.2.1 Research problem . . . 125
6.2.2.2 Ideas for potential solutions . . . 126
6.2.3 FPGA-based execution platforms . . . 127
6.2.3.1 Research problem . . . 127
6.2.3.2 Ideas for potential solutions . . . 128

List of Figures

2.1 Camera-only and cloud-only solutions 12
2.2 Three approaches to deep-learning inference in IoT environment with an edge and a host platform 13
2.3 The generic teacher-student framework for knowledge distillation 20
2.4 Illustration of intermediate results of a DNN model 22
2.5 The bottleneck unit embedded with a non-differentiable lossy compression 25
2.6 Heterogeneous recognition difficulty of test instances 29
2.7 Illustration of the first five layers of an MSDNet with three scales . 31
2.8 Left: for a given test instance, the green and red circles represent the correct and incorrect output predictions respectively 32
2.9 Decision gates (d-gate) integrated directly into deep neural networks 33
2.10 A conceptual overview 35
2.11 The collaborative cloud-edge computing framework 41
2.12 Reuse data management in 42

3.1 The overall process of Cut, Distil and Encode (Cut, Distill, Encode (CDE)) 45
3.2 The size of output intermediate data and number of parameters of each layer for (a) VGG16 and (b) MobileNetV1 48
3.3 The distributed DNN architecture 50
3.4 Visualisation of the output latent representation from: (a) the hint layer, (b) the guided layer, and (c) the auto-encoder's output . . . 53
3.5 Accuracy of VGG16 when splitting the network at layers 55
3.6 Accuracy of MobileNetV1 when splitting the network 56

3.7 The impact of communication cost on accuracy when we split VGG16 at layer 7 and MobileNetV1 at layer 28 58

4.1 Comparison between different early-exit methods 68
4.2 System overview 71
4.3 Same class images output by different exits 75
4.4 Distribution of confidence scores at each exit 77
4.5 Percentage of images exiting at each exit with different workload sharing methods 85
4.6 Time/Accuracy graph for different methods 88

5.1 Energy consumption estimation flow. 96
5.2 Architecture components energy consumption 97

List of Tables

2.1 Some off-the-shelf microcontroller development platforms with their SRAM and flash storage constraints 16
2.2 Memory size required at the edge when splitting MobilenetV1 and VGG16 at different layers 23

3.1 The size of intermediate data when splitting MobileNetV1 and VGG16 at different layersk 49
3.2 The accuracy of VGG16 and MobileNetV1 for different memory sizes at the edge with 3G (200 KB) communication constraint . . 59
3.3 The accuracy of VGG16 and MobileNetV1 for different memory sizes at the edge with LoRa (25 KB) communication constraint . . 60
3.4 Comparison of CDE with E-JPEG and E-AE 63
3.5 The trade-off between accuracy and memory size at the edge between our CDE and CE-AE 64

4.1 True Positives (TP) and False Positives (FP) scores for VGG16 . . 76
4.2 F1 scores and accuracies for VGG16 76
4.3 Memory size of the early-exits' output when using different architectures and training methods 83
4.4 Accuracy of the early-exits when using different architectures and training methods 83
4.5 Percentage of images taking each early exit with linear programming workload sharing 86
4.6 Overall error rate and the overall average execution time per image 86

5.1 Processor's activities counting 107
5.2 Memory's activities counting 108

5.3 Energy consumption for a MIPS R3000 110
5.4 Energy consumption for an Infineon CY62146ESL MoBL SRAM 111
5.5 Energy consumption per byte for Microchip RN2903 and ZG2100M network controllers 112
5.6 Dynamic energy consumption for each EE per image (in μ J) . . . 113
5.7 Available energy and energy sources for the cloud and each edge . 114
5.8 Images taking each early exit according to the initial LP resolution 114
5.9 Images exiting at each EE and corresponding energy consumption with fixed thresholds 117
5.10 Images exiting at each EE and corresponding energy consumption with dynamic threshold 117
5.11 Performance sensitivity to the presence/absence of permanent energy sources 118
5.12 Energy consumption (in MJ) of Cameras + Cloud approach 120
5.13 Energy consumption (in MJ) of Cloud-only without EE approach . 120
5.14 Energy consumption (in MJ) of Cloud-only with EE approach . . 120

List of Acronyms

AE Auto encoder

AI Artificial intelligence

ALU Arithmetic-logic unit

CDE Cut, Distill, Encode

CE Cut, Encode

CNN Convolutional neural network

DNN Deep neural network

DTC Dynamic threshold configuration

EE Early-exit

FC Fully connected

FP False positive

FSM Finite-state machine

FT Fixed threshold

GPU Graphics processing unit

HD High definition

HT Hint training

IoT Internet of things

JL Joint learning

KD Knowledge distillation

LoRa Long range wide area

LP linear program

ML Machine learning

PE Processing element

RAM Random-access memory

RISC Reduced instruction set computer

RNN Recurrent Neural Network

SL Separate learning

SRAM Static random-access memory

TP True positive

UHD Ultra high definition

1 | Introduction

1.1 Context and motivation

In the past decade, we have witnessed the increasing development of edge computing operating on a wide range of devices including mobile phones, cameras, home intelligence devices, robots, drones, and sensor networks. These devices, referred to as *edge devices*, are physical devices equipped with sensing, computing, and communication capabilities. They typically operate continuously, collecting a large amount of data including images, videos, audio, and sensor data with the ultimate goal of providing actionable information to the user or system.

Concomitantly to the emergence of the edge computing paradigm, deep learning has become one of the most dominant data analytic approaches due to its capability to achieve impressively high accuracies on a variety of key computing tasks such as classification [Alex et al., 2012], scene understanding [Bolei et al., 2014], speech recognition [Hinton et al., 2012], and face detection [Yaniv et al., 2014]. Some of these tasks need intensive computation especially when image and video data captured from camera(s) are input to the predictive model. Indeed, neural networks are known to be very expensive in terms of memory, computation and energy, making their execution by edge devices challenging [Zhang et al., 2020].

In traditional systems like video surveillance, these tasks are offloaded to the cloud server by directly streaming video to the cloud [Google, 2016, Lovejoy, 2015]. The cloud servers then automatically process the collected data by employing machine (deep) learning algorithms. This solution has limitations, most importantly high latency and network load due to limited bandwidth, impacting the performance of real-time applications. In particular, in environments where the network

is severely limited, this solution is not viable. This is the case in several application scenarios, such as camera networks deployed in tunnels to detect events of interest (fires, accidents) [Torres-Zapata et al., 2019]. Another example is smart agriculture, where the use of cameras makes it possible to optimize harvests (monitor growth, detect parasites) [Shadrin et al., 2020, Segalla et al., 2020] or even monitor food distribution among livestock [Monteiro et al., 2021]. The cameras are positioned in remote fields where there is poor network availability, which can especially be the case in developing countries or rural areas. Another example is the use of cameras to capture observations of animals in nature reserves located in remote areas [Zualkernan et al., 2022]. The common solution is to collect images onto memory cards using camera traps, which are then processed several months later. This does not make it possible to learn about events of interest in real time (presence of a rare species or poachers).

Moreover, sending large amounts of sensor data is very energy-expensive for limited battery powered end devices, making this solution impractical for long period operations. Other issues including leaking user privacy (e.g., raw data collected at edge devices may contain sensitive and private information about individuals), and suffering from unpredictable delays.

In the last few years, continuous advances in microelectronics and integration technologies have made it possible to increase the computational power of the mobile platform, leading to the edge computing paradigm. These are systems where Machine learning (ML) algorithms are no longer deployed on powerful cloud servers but they are embedded directly on the edge device (e.g. cameras) [Lin et al., 2020, Jouppi et al., 2017, Sandler et al., 2018, Liu et al., 2016]. Such solutions certainly eliminate congestion problems in the networks, but they are greatly limited by the amount of computation and storage resources that can be included on the camera, for power and cost reasons. These resources are still very

limited and significantly lower than what we could find on the cloud servers. To that end, model compression techniques are typically required to enable efficient execution on the edge although this usually compromises accuracy.

Since then, some research has examined approaches which distribute the inference of machine learning algorithms between the edge (camera, mobile robot, etc.) and the cloud [Kang et al., 2017, Emmons et al., 2019, Ashish et al., 2015, Jong Hwan et al., 2018]. This has the advantage of reducing the communication burden (e.g. bandwidth) since the raw data are directly processed on the edge by a subset of the algorithms while at the same time maintaining the model accuracy by executing the remaining, resource-intensive parts of the algorithms, on the cloud.

The distributed computing scenario has resulted in a research problem that is both complex and of great practical interest. In the context of distributed deep learning for example, it consists of determining a 'good' partitioning of neural network layers between the edge and the cloud while respecting some constraints such as the limited availability of computing resources at the edge, the limited bandwidth usage, the number of frames processed per second, and the tolerable limits in terms of loss of precision. An additional difficulty is the data amplification problem: most of the time, the output data of the intermediate layers of a Deep neural network (DNN) (Deep Neural Network) are much larger than the original image. Hence, there is a need to introduce into these distributed systems compression techniques offering guarantees for both reducing the size of the transmitted data and fit the memory of the edge, while minimizing the loss of precision.

Moreover, the distributed inference of a DNN between the cloud and multiple nodes has hardly been explored, whereas in practice, it is becoming more common to find multi-camera networks for functional needs (e.g. monitoring several places or capturing views from different angles) [Aghajan and Cavallaro,

2009, Ryan et al., 2014, Yao et al., 2020a]. More generally, distributed computing systems have become common, especially in the Internet of things (IoT) (Internet of Things) context [Chien et al., 2015, Remagnino et al., 2004, Ahmed et al., 2020, Patel, 2019]. These are characterized by short-range, high-speed links (e.g. WiFi) and long-range, low-speed links (e.g. cellular or Long range wide area (LoRa)). In this context, the local distribution of a DNN is of great interest, especially when the network access is poor and the computation nodes have limited capacities. Hence, an efficient approach must be developed to distribute the DNN inference according to the time, precision and energy constraints.

1.2 Research challenges

Recent developments of DNNs for image processing show promising results. However, there still are a number of drawbacks leading to the degradation of the performance of such models, particularly when they are deployed on Cloud / Edge type platforms and used on cameras embedding limited hardware resources (micro controllers and small size memory for example).

With the technological advances of recent years, most modern video surveillance systems consist of a set of nodes with/without video sensor, each embedding limited computation and storage elements, and a geographically distant cloud, equipped with a significant computation power. All these elements are networked with various speeds and protocols (LoRa, 3G, 4G, Wifi...). The real scientific challenge nowadays is therefore not only to make sure that the DNNs are executed as quickly as possible, but to use all the available resources in the cloud / edge platform in the best possible way, while respecting the constraints and limitations in terms of computing, storage, energy and network resources, for fast execution without loss of precision.

Several existing works propose methods for partitioning and assigning DNN layers between the cloud a single edge device [Kang et al., 2017, Chien-Chun et al., 2018]. Other ones include solutions to compress the layers of a DNN [Yao et al., 2017, Gupta et al., 2015, Saputra et al., 2019, Romero et al., 2015] or inter-layers data to target execution on edges with limited resources [Emmons et al., 2019, Peisong et al., 2018]. Moreover, some authors have shown that it is even possible to adapt the amount of computation of a DNN to the difficulty of each input image [Teerapittayanon et al., 2017b, Li et al., 2019, Laskaridis et al., 2020a]. Separately and each taken on its own, most of these approaches have been proven to be effective on various types of DNNs, application functions and architectures. However, to the best of our knowledge, few existing works have explored the distribution of a neural network between the cloud and several computing elements at the edge. Moreover, they only optimise accuracy [Teerapittayanon et al., 2017a]. The cloud / multi-edge combination with limited camera resources, plus limited communication network bandwidth and heterogeneous input image input remains unexplored for the distributed and optimized execution of DNNs.

Therefore, the research question that interests us is: *Given a cloud / edge type environment with limited resources on computing nodes and network, how to take advantage of all the available resources to efficiently execute a DNN while optimizing time, accuracy, and respecting resource constraints?*

This research question implies several challenges :

- Partitioning a DNN inference between the edge and the cloud implies a trade off between two competing aspects : memory size at the edge and size of the intermediate data. Indeed, the size of intermediate results generated from each layer decreases from early layers to later layers [Zhang et al., 2020]. Therefore, splitting the DNN at an early layer implies less memory

requirement at the edge but larger intermediate data. To limit this effect, previous works proposed to split the network in the last layers, just before the computationally intensive fully connected layers [Kang et al., 2017]. This solution, where most of the network execution is left to the edge device, is not feasible for limited devices such as Micro-Controller Units (MCUs).

- Another problem that can significantly degrade performance lies in the use of identical processing for all input samples of a DNN. For instance, in image classification applications, some input images may be completely devoid of any element of interest, while others may suffer from bad illumination, complex poses, zoom, etc. The degree of complexity of the prediction for these different images can vary significantly. Therefore, processing should take into consideration the type of input. Early-exit is a kind of technique to terminate a pre-specified computation at an early stage depending on the input samples. Integrating early exits in the DNN is a promising approach to taking into consideration the heterogeneous complexity of input images. Moreover, the introduction of early exits allows to take advantage of the multiple computing elements of the target platform to optimize execution time, by including early-exits (EE) on each node. Early-exit (EE) strategies for terminating computation of 'easy' classes have been proposed, but these consider a single computing element at the edge level [Laskaridis et al., 2020a, Zeng et al., 2019, Wolczyk et al., 2021].

- Finally, the optimised execution, in terms of energy consumption, of a DNN on a single edge and a cloud is already a complex scientific challenge, but the extension of the model for collaborative multi-edge systems is of much greater complexity. This difficulty is mainly due to the heterogeneity of the potential available resources in each node in the platform. Furthermore, the exact estimation of energy consumption is an extremely tedious task

requiring the development of prototypes corresponding to all the hardware architectures considered in the approach and the use of electronic probes. This is unfortunately unsuitable in the context of our approach, because the latter does not target a single and specific hardware architecture or a single particular application.

1.3 Research objectives

The objectives of this research can be summarized as follows:

1. A comprehensive review of the state of the art approaches for DNN model compression, DNNs partitioning on a cloud/edge platform, and dynamic data dependent processing in image/video applications.
2. Development and evaluation of a novel and effective DNN partitioning and compression approach for distributed execution on a cloud / single-edge resource constrained system.
3. Extension of the proposed approach for supporting multiple edge devices and input image dependent processing.
4. A fast and dynamic high level estimation and optimisation approach for multi-edge platforms with limited available energy.

1.4 Contributions

The contribution of this thesis is the design of a novel distributed DNN architecture and optimization strategy for accurate and efficient inference on constrained cloud/edge environment. The details of the contributions are described as follows :

- **Contribution 1:** We introduce a novel partitioning approach for DNN inference between a single edge and the cloud, by considering simultaneous optimization of both the memory usage at the edge and the size of the data to be transferred over the wireless link. We perform our experiments on two different network architectures, MobileNetV1 and VGG16. The proposed approach makes it possible to execute part of the network on very constrained devices (e.g., microcontrollers), and under poor network conditions (e.g., LoRa) whilst retaining reasonable accuracies. Moreover, we show that the choice of the optimal layer to split the network depends on the bandwidth and memory constraints, whereas prior work suggests that the best choice is always to split the network at higher layers.

- **Contribution 2:** We propose an image dependant processing approach, based on early exits, for time optimization of Convolutional neural network (CNN)s (Convolutional Neural Networks) inference in constrained cloud- multi-edge environments. By including early-exits and bottlenecks in the neural network, the solution we propose makes it possible to distribute computation according to time and accuracy requirements. This approach includes a workload sharing policy based on linear programming to distribute the execution between the different nodes. The workload sharing between the different early-exits is determined at compile time (offline), and it is adjusted at run time (online) with a method that we named Dynamic threshold configuration (DTC). In scenarios where the communication with the cloud is very limited (e.g. LoRa), it allows a 80% improvement in average latency compared to a traditional cloud solution. Moreover, we show that it allows a better accuracy-latency trade-off than the fixed-threshold workload sharing policy.

- **Contribution 3:** We propose a dynamic approach for energy consumption optimization targeting cloud/multi-edge execution platforms. First, we develop a solution to make energy consumption estimates with acceptable accuracy, while supporting various architectures and applications. This methodology remains fast enough to operate on application models without resorting to the development of physical prototypes. We then use this approach to optimise energy consumption in distributed DNN inference. This is particularly important when the targeted execution platforms feature one or more battery-operated edges, where power is the critical resource. Our optimisation flow is based on dynamic linear programming and is reactive to energy-related events during run time. The results show that this approach allows to identify the optimal configuration that minimizes energy consumption while maintaining the desired accuracy level.

1.5 Thesis structure

The rest of this thesis is organised as follows:

Chapter 2 presents the existing works related to the issues raised in the thesis, namely DNN inference distribution, DNN compression, and image-dependant DNN processing.

In Chapter 3, we propose a novel DNN partitioning approach for a cloud/single edge platform. This approach aims at finding the best partition point according to both memory and bandwidth constraints, while optimising accuracy.

Chapter 4 extends the proposed approach to allow an image-dependant and multi-edge processing. We achieve this by introducing early-exits in the DNN architec-

ture. We also develop a workload sharing strategy to distributed the input images between the different nodes and the cloud while optimising accuracy and time.

Chapter 5 focuses on energy-constrained environments where the edge nodes are battery-powered. We propose a dynamic workload-sharing strategy to optimise energy consumption and accuracy. This includes an energy-consumption estimation approach.

Finally, Chapter 6 concludes this work, with a discussion about the limitations of the proposed approach and perspectives for future work.

1.6 Publications

The contributions of this thesis have led to the following publications.

1. M. Sbai, M. R. U. Saputra, N. Trigoni and A. Markham, "Cut, Distil and Encode (CDE): Split Cloud-Edge Deep Inference," 2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON).

2. M. Sbai, N. Trigoni and A. Markham, "Multiple early-exits strategy for DNN inference", 2023 19th Annual International Conference on Distributed Computing in Smart Systems and the IoT (DCOSS-IoT).

3. M. Sbai, N. Trigoni and A. Markham, "Energy consumption estimation and optimisation for distributed DNN inference", under submission process.

I am lead author on these publications and chapters in this thesis and I conceptualised, designed and implemented these techniques. Risqi Saputra assisted in Chapter 3 with network distillation. Professors Trigoni and Markham supervised and guided the research program.

2 | Background

2.1 Introduction

It is widely known that processing image and video data by DNNs requires intensive computation [Hauswald et al., 2015]. Traditional cameras are typically not equipped with enough computational resources for processing large video data, especially when latency and energy have to be respected. A common solution to this problem is to transmit and offload all the data to be processed by the cloud as seen in [Lovejoy, 2015] and [Google, 2016]. This data must be compressed locally on the camera beforehand and then sent over a channel with limited bandwidth. This solution has limitations, most importantly high latency and network load due to limited bandwidth, impacting the performance of real-time applications. Moreover, sending large amounts of sensor data is very energy-expensive for limited battery powered end devices, making this solution impractical for long period operations. In addition to this disadvantage, the appearance of artefacts on compressed images (videos) involves risks of losing their semantic content, and in any case considerably reducing the performance of the DNNs [Emmons et al., 2019] (fig. 2.1). Other issues including leaking user privacy (e.g., raw data collected at edge devices may contain sensitive and private information about individuals), and suffering from unpredictable delays.

With the increasing development of edge-devices, other approaches have emerged. The three main solutions nowadays are cloud-only inference, edge-only inference, and distributed inference, as shown in figure 2.2. In this chapter, we provide an overview of distributed computing hierarchy, particularly seen from the perspective of DNN deployment, with the edge-only inference (Section 2.2) and the cloud/single-edge inference (Section 2.3). We also present the research works that

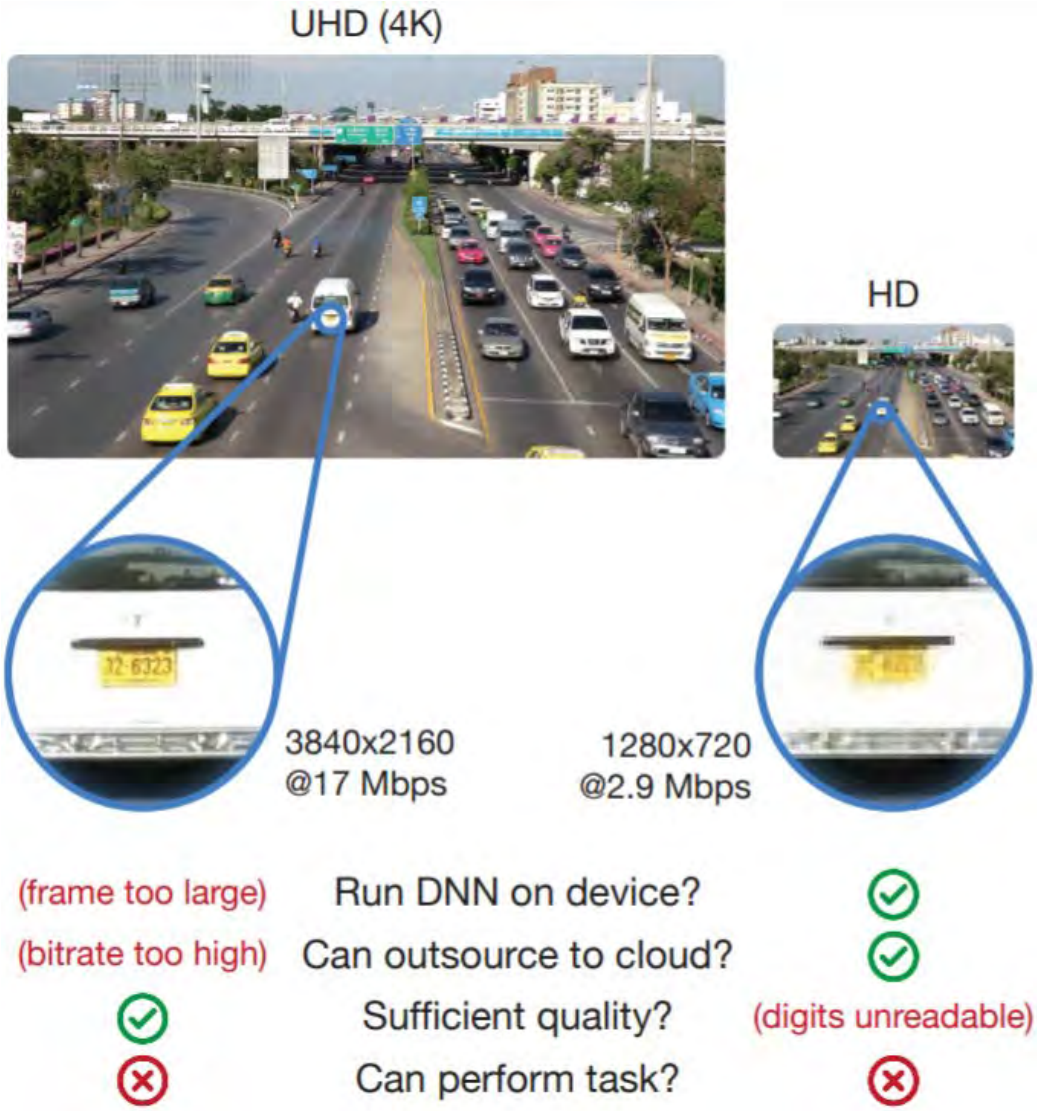


Figure 2.1: Camera-only and cloud-only solutions. Image resolution plays an important role: The 4K Ultra high definition (UHD) image (left) provides sufficient quality to detect the license plate, but is too big to be processed on camera or sent over a network. The compressed HD version (right) is small enough to be processed on device or sent to the cloud, but leads to inaccurate reading of the license plate. [Emmons et al., 2019]

have been done on dynamic inference in deep neural networks (Section 2.4), be-

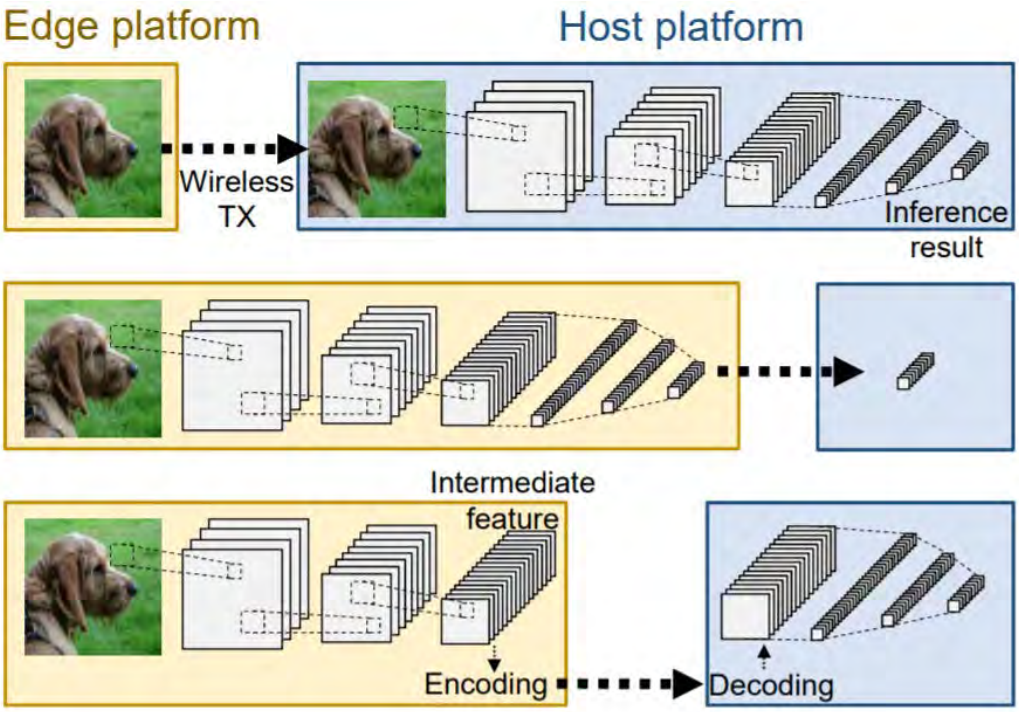


Figure 2.2: Three approaches to deep-learning inference in IoT environment with an edge and a host platform (top) Entire inference at the host using images transmitted by the edge, (middle) entire inference at the edge that transmits the end output to the host, and (bottom) the proposed partitioned inference where the edge processes inference up to an intermediate layer, whose features are encoded and transmitted to the host for inference of the rest of the network. [Jong Hwan et al., 2018]

fore exposing the few works that have been done on distributed cloud/multi-edge DNN inference (Section 2.5).

2.2 Edge computing

The edge-computing approach consists in performing the entire computation on the edge devices (e.g. mobile phone, camera). The edge platforms are heterogeneous in terms of resources (Section 2.2.1). As a consequence, several specific DNN architectures have been designed, targeting different types of edge devices

(Section 2.2.2). More generally, several DNN compression techniques can be used to reduced the size of a deep neural network (Section 2.2.3).

2.2.1 Heterogeneity of edge devices

The edge computing technique is motivated by the very high cost generated by the transfer of a mass of data from the edge to the cloud. Deploying a deep learning model on the edge may be a better alternative for applications where bandwidth is scarce, low latency is required, and/or security is an issue. For example, camera traps are typically deployed in remote areas where network bandwidth and speed are limited, and therefore sending all captured footage to the cloud is potentially slow and expensive. This is particularly relevant because many captured images are blank or ghosted and provide no interesting or important information. IoT cameras, such as dash-mounted cameras, drone cameras, Google glasses and phone cameras are already widespread in our lives and are used for several applications such as traffic regulation or surveillance.

Nevertheless, executing DNNs on the edge devices remains a challenge. Conventional DNN architecture (e.g. AlexNet, VGGNet, ResNet, etc.) typically require specialised hardware (such as Graphics processing unit (GPU) - Graphics processing unit) to speed up calculations and operate in real time on High definition (HD) videos [Jouppi et al., 2017]. This is highly power consuming and not feasible for applications that continuously need to perform computations. These difficulties drastically increase for edge devices that have extremely limited resources such as low-end IoT devices, which may not be capable of executing an entire DNN model locally.

Indeed, there are various types of edge devices, with heterogeneous capacities. For example, modern smartphones such as iPhone 13 are equipped with 3.23 GHz

hexa-core CPU, a GPU, a 4 GB Random-access memory (RAM) (Random-access memory) and 512 GB storage [GSMArena, 2023a]. Samsung Galaxy S20 has 2.84 GHz octa-core CPU, a GPU, a 8 GB RAM and 128 GB storage [GSMArena, 2023b]. Another example is smart cameras equipped with single-board computers such as Rasperry Pi [RaspberryPiLtd, 2023] (1.5 GHz quad-core CPU, GPU, 8 GB RAM, micro-SD storage) and NVIDIA Jetson [NvidiaCorporation, 2023] (1.43 GHz quad-core CPU, GPU, 4 GB RAM, micro-SD storage). These types of devices must be frequently alimented in energy.

Other edge devices are much more restricted. This is particularly the case of microcontrollers (MCUs), which are not equipped with GPUs, and whose memory capacity is 2-3 orders of magnitude smaller even than mobile phone [Lin et al., 2020]. MCUs have the advantages of being cheap, widespread and geared towards energy-efficient workloads. They are used in battery-powered cameras for various applications, such as traffic regulation [Alessandrelli et al., 2012], person detection [Chowdhery et al., 2019], or image classification [Lin et al., 2020].

These devices are often equipped with microcontrollers (MCUs), which have the advantages of being cheap, widespread and geared towards energy-efficient workloads. However, an MCU typically consists of a low-frequency processor and only several hundred kilobytes of on-chip memory [STMicroelectronics, 2023a], as shown in Table 2.1. Typical microcontrollers have extremely limited on-chip memory (100–320 KB Static random-access memory (SRAM)) and flash storage (256 KB–2 MB). The entire DNN model with its weight parameters and code has to fit within the small memory budget of flash storage. As a comparison, for a classification task, VGG16 [Qassim et al., 2017] has a size of 553.16 MB and even a model like MobileNetV1 [Howard et al., 2017] which is specifically targeted to mobile platforms has a size of 17.3 MB, which clearly exceeds the

memory capacity of MCUs.

Table 2.1: Some off-the-shelf microcontroller development platforms with their SRAM and flash storage constraints [Chowdhery et al., 2019]. Note that we list the microcontrollers with at least 250 KB SRAM, though other examples such as Cortex M0 only have 20–30 KB SRAM.

MCU platform	Processor	Frequency	SRAM	Flash
ATSAM4E16E [Microchip, 2023]	Cortex-M4	120 MHz	128 KB	1 MB
FRDM-K64F [NXP, 2023b]	Cortex-M4	120 MHz	256 KB	1 MB
STM32 F723 [STMicroelectronics, 2023c]	Cortex-M7	216 MHz	256 KB	512 KB
Nucleo F746ZG [STMicroelectronics, 2023b]	Cortex-M7	216 MHz	320 KB	1 MB
NXP i.MX1060 [NXP, 2023a]	Cortex-M7	600 MHz	1 MB	External

2.2.2 Specific DNN architectures for edge computing

To be able to run on an end-device, neural networks must be optimized. Many recent works propose stripped-back neural networks capable of running on the edge, so called TinyML [Ray, 2022]. These types of neural networks have been developed for various types of applications, ranging from speech recognition [Wong et al., 2020a] to hand gesture recognition [Zhou et al., 2021]. Several works have proposed neural networks optimized for image classification, as presented below.

Tensorflow Lite and Tensor RT are two frameworks optimized for mobile devices and embedded devices [Prado et al., 2020]. However, optimizing a neural network for edge-inference can impact inference speed, power consumption, and performance. For example, Mathur and Khattar [Mathur and Khattar, 2019] compared the TensorFlow and TensorFlow Lite versions of their model and noticed that the ResNet-50 took about ¼ of the time to classify a single image while using twice the power. Similarly, Ramos-Arredondo et al. [Ramos-Arredondo et al., 2020] compared MobileNetV2 to a TensorFlow Lite version. They noticed that the TensorFlow Lite version had more frequent false detections.

In addition to using the standardized optimization frameworks, custom optimizations can also be used. For example, a pruned CNN architecture that resulted in a smaller, less complex network that had the same performance as the original network was proposed by Rohilla et al. [Rohilla et al., 2020]. By doing so, they were able to reduce the number of parameters in a five-layer CNN model by 94% (from 40,000 to 2,400 parameters). Curtin and Mathews [Curtin and Matthews, 2019] used an augmented Raspberry Pi based camera system with a deep learning image recognition model to detect wildlife. The system used a simple CNN with two convolutional layers to classify images into one of three classes: snow leopard, human, and other. The accuracy obtained was 74%n and the system has an availability of only 10 hours before discharging the battery. Similarly, Thomassen [Thomassen, 2017] deployed MobileNet on a Raspberry Pi. The model achieved 81.1% accuracy on the COAT dataset with a model size of 17 MB. The Raspberry Pi was able to analyze 2.1 million images using a car battery.

Neural networks targeting microcontrollers are mostly specific to limited applications such as binary classification (e.g. Visual Wake Word [Chowdhery et al., 2019]). SSDLite [Sandler et al., 2018], a mobile version of Single Shot Detector (SSD) [Liu et al., 2016], requires 15 times more computation time to process a high resolution video even when it is optimised for edge devices [Emmons et al., 2019]. MCUNet [Lin et al., 2020], despite a remarkable compression rate which enables ImageNet inference on microcontrollers, can only reach around 70% top1 accuracy. This is also the case of AttendNets [Wong et al., 2020b], which uses 4.17x less parameters and 16.7x minimized weight memory when compared to MobileNet-V1, but has an accuracy of 71.1 % precision on the Imagenet50 benchmark.

2.2.3 DNN compression techniques

A natural solution is to perform model compression in order to speed up the computations without signicantly decreasing the performance of the model. In the last few years, great progress has been made in the area of DNN compression. Recent advanced techniques for squeezing and accelerating DNNs are generally classified into four groups: network pruning, quantization, decomposition, and knowledge distillation (Knowledge distillation (KD)).

Network pruning removes redundant and uninformative weights in the network. Weight pruning can be performed using a dropout-based method [Yao et al., 2017, Molchanov et al., 2017] or a magnitude-based method [Han et al., 2015, Guo et al., 2016]. Pruning can be applied on connections between neurons [Han et al., 2015,Han et al., 2017] or on convolutional filters [Luo et al., 2017]. This approach promises a reduction of the parameters without affecting the accuracy, although it generally requires more processing steps [Han et al., 2017].

Quantization compresses the network parameters in order to reduce the size of the weight representation [Gupta et al., 2015, Vincent et al., 2011]. Quantization can be applied by using a 16-bit or 8-bit representation as proposed by [Gupta et al., 2015, Vincent et al., 2011]. In an extreme case, a representation in 2-bit (binary network) can also be used as shown in [Courbariaux et al., 2015, Hubara et al., 2016]. By restricting weights to only two values, binary networks can drastically reduce computing time and memory consumption, but the accuracy is also significantly reduced [Cheng et al., 2018].

The decomposition technique reduces the complexity of the network by introducing low-level constraints on the network weights (low-rank decomposition) [Rigamonti et al., 2013, Bhattacharya and Lane, 2016]. Most approaches factorize a

fully connected or convolutional layer using matrix/tensor decomposition techniques such as SVD or CP (Principal Components) decomposition. By using this factorization, the number of multiplications is considerably reduced. However, the obtained compression ratio is generally lower than that obtained by the pruning-based approach [Saputra et al., 2019]. Moreover, the low-level constraint imposed on the network may have a negative impact on the network performances.

Knowledge Distillation (KD) technique is an approach that transfers knowledge from a large network of neurons, called teacher, to a small network called student [Hinton et al., 2012]. The key idea of knowledge distillation is to transfer the knowledge from the large teacher model to the small student model. Generally, the teacher-student framework for knowledge distillation is shown in fig.2.3 We find encouraging results in the literature with networks of students having finer and deeper architectures (e.g. Fitnet) [Romero et al., 2015, Saputra et al., 2019]. This approach is based on step wise training by training the first part of the student before training the entire network. The authors used an intermediate representation of the teacher model as a hint in addition to the output layer, which improved performance when distilling knowledge from deeper teacher networks. Yim et al. [Yim et al., 2017] applied knowledge distillation to the ResNet architecture by minimizing the L2 loss of Gramian [Gatys et al., 2015] feature matrix in the ResNet modules between teacher and student.

2.2.4 Discussion

Edge computing is an interesting alternative to the traditional cloud paradigm. It has the advantage of not being dependent on the network condition, which enables a faster inference when network conditions are poor. However, this approach is limited by the edge-devices computing and power capacities. On very constrained devices such as MCUs, reducing the model size results in lower ac-

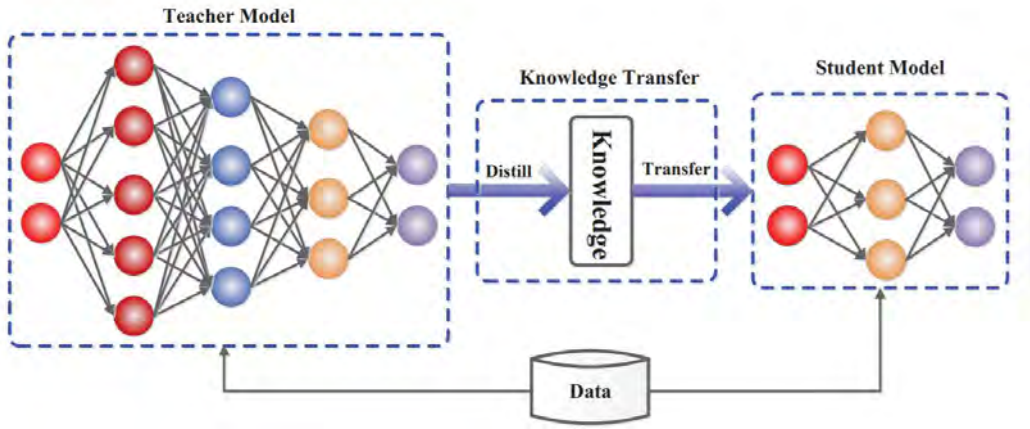


Figure 2.3: The generic teacher-student framework for knowledge distillation. [Gou et al., 2021]

curacy. This requires compressing large, highly accurate models through various complex techniques in order to obtain a compact model on the device, albeit with lower accuracy. Aside from these resource constraints, there may be other restrictions preventing full execution of neural networks at the edge, such as trade secrets: an analytics service or an app provider might not be keen on sharing its valuable and highly tuned model. Therefore, it is not always possible to assume local processing is a viable solution even if the task duration, memory, and processing requirements are not critical for the user, or the task can be performed when users are not actively using their devices (e.g., when the device is being charged overnight) [Osia et al., 2020].

2.3 Cloud/Single-edge DNN partitioning

A compromise between performing computation entirely on the edge or only on the cloud has just emerged in the last years [Kang et al., 2017, Emmons et al., 2019, Ashish et al., 2015, Jong Hwan et al., 2018]. These techniques involve investigating computational partitioning strategies that efficiently leverage units in

the cloud and on the node (camera, mobile device, etc.) to achieve low latency, low power consumption, and high data center throughput. Towards this goal, several key aspects must be taken into account. The first one is the size of the intermediate data, which is often high because of the data amplification issue (Section 2.3.1). To address this, some works have proposed to compress the intermediate data (Section 2.3.2). The second key aspect to take into account is the memory capacity of the edge device because the model must fit the memory of the edge device. Few works have proposed solution to optimise the neural network layers size at the edge (Section 2.3.3).

2.3.1 Data amplification issue

In deep neural networks, the sizes of intermediate results generated out of each layer decrease from lower layers to higher layers (fig. 2.4). As a result, partitioning at lower layers would generate larger sizes of intermediate results, which could increase the transmission latency [Zhang et al., 2020]. Moreover, there is another important difficulty: the *data amplification* problem. This problem refers to the fact that the output data of the intermediate layers of a DNN are typically much larger than the original image.

To limit this effect, previous works proposed to split the network in the last layers, just before the computationally intensive fully connected layers. Kang et al. [Kang et al., 2017] present a solution for offloading the computation of the conventional DNNs from 8 legacy architectures (e.g., AlexNet, VGG) between the edge and the cloud. These DNNs, in their designs, offer a natural sharing point at their Fully Connected (FC) layers. In that sense, the distributed model will not suffer from data amplification. Power consumption and latency are the two mains parameters used to evaluate the performance of their technique. The need for data compres-

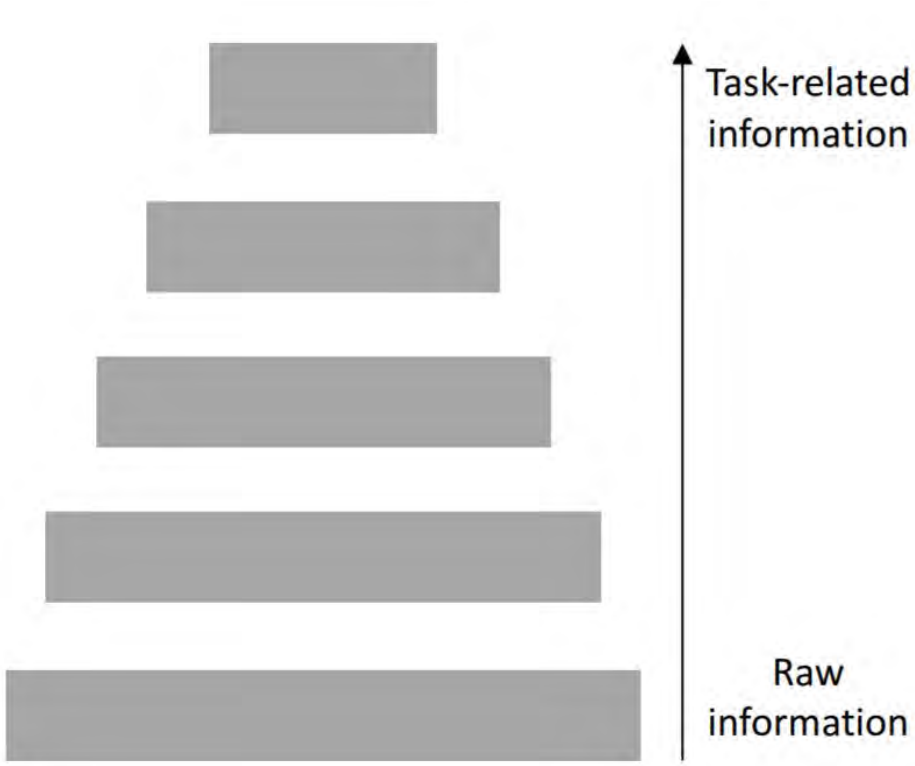


Figure 2.4: Illustration of intermediate results of a DNN model. The size of intermediate results generated out of each layer decreases from lower layers to higher layers. The amount of information generated out of each layer also decreases from lower layers to higher layers [Zhang et al., 2020].

sion prior to transmission has not been addressed, so the performance in terms of accuracy remains unchanged.

Xu et al. [Xu et al., 2020a] propose a collaborative cloud-edge computing framework to optimally deploy neural network model into the edge nodes. An adaptive task scheduling algorithm is designed to adaptively optimize the task assignment by using the improved ant colony algorithm. The simulation results show that the framework can reduce time delay and energy consumption, and improve task accuracy in comparison to the cloud-only and to the edge-only scenarios. However,

the neural network is processed blindly and no optimisation is done to remedy the data-amplification issue and the memory usage at the edge.

This solution, where most of the network execution is left to the edge device, is not feasible for devices such as MCUs. Table 2.2 shows that the memory sizes of VGG16 and MobileNetV1 when splitting at different layers are generally bigger than the capacity of MCUs.

Table 2.2: Memory size required at the edge when splitting MobilenetV1 and VGG16 at different layers.

Model	Layer	Mem. size
MobileNetV1	28	21KB
MobileNetV1	41	662KB
VGG16	7	1MB
VGG16	11	7MB
VGG16	15	29MB

More recent works takes data amplification into account by proposing intermediate data compression.

2.3.2 Intermediate data compression

A possible approach to alleviate the data amplification problem is by compressing the data produced by the last layer of the network, but this must not cause an accuracy drop. Hence, some works suffer from the phenomenon of data amplification even after reducing the size of the intermediate data. Ko et al. [Jong Hwan et al., 2018] have introduced introduced a lossless compression (run-length encoding combined with Huffman encoding) on the data sent to the cloud, which allows a compression ratio of 3 to 10. They also proposed to compress the intermediate data before sending it to the cloud using the standard JPEG encoding with a loss

of accuracy of only 1%. However, this data is still bigger than the original video stream.

The results of these attempts to split the DNN [Jong Hwan et al., 2018, Xu et al., 2020a] converge to the execution of the whole convolutional part on the node and the fully connected part on the cloud. Indeed, the convolutional part suffers from the phenomenon of data amplification: it is only from the first fully connected layer that the data will be reduced approximately to the input size or less. It follows from these techniques that the nodes should be well equipped in terms of computing hardware in order to execute all the convolutions of the network.

In order to partition the DNN more efficiently, Emmons et al. [Emmons et al., 2019] present a compression technique composed of three steps: quantization, factorization and coding. This compression technique allows one to obtain intermediate data which is up to 7x smaller than the original video stream. Despite the interesting results obtained via this approach, the memory capacity at the edge is not considered to find the optimal partitioning point.

Eshratifar et al. [Eshratifar et al., 2019] present a partitioning scheme that creates a bottleneck in a neural in order reduce the communication costs of feature transfer between the mobile and cloud(fig. 2.5). This method seems generic and might apply to different DNN architectures and hardware platforms. Its main idea consists in selecting the best partition point for the minimum end- to-end latency and/or mobile energy consumption at runtime. It presents some interesting results in terms of latency and energy consumption on the node side. However this work do not give any performances related to the accuracy of the obtained distributed models. Moreover, to find the best partition, this work does not take the edge capacities in terms of memory, whereas it is an important constraint with very limited edges such as MCUs.

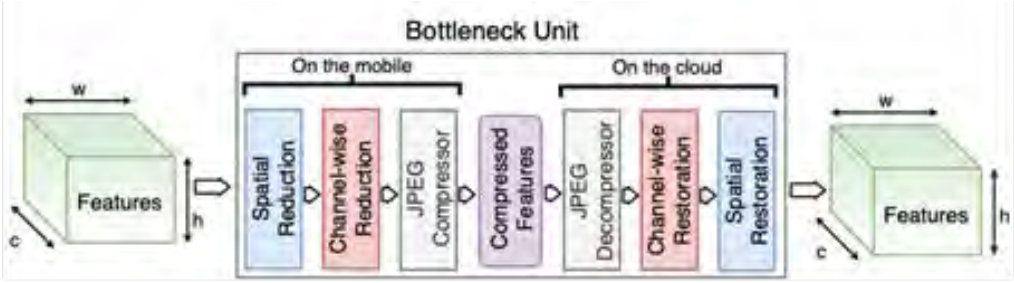


Figure 2.5: The bottleneck unit embedded with a non-differentiable lossy compression [Eshratifar et al., 2019]

Choi et al. [Choi et al., 2020] presents a prediction scheme called Back-and-Forth (BaF) prediction, developed for deep feature tensors, allowing to reduce tensor size and improve its compressibility. This approach seems giving interesting results in terms of compression rate comparing to standard compression algorithms. Shao et al. [Shao and Zhang, 2020] introduce an end-to-end architecture, named BottleNet++, that consists of an encoder, a non-trainable channel layer, and a decoder. The encoder and decoder essentially implement joint source-channel coding via CNN. By exploiting the strong sparsity and the fault-tolerant property of the intermediate feature in a DNN, BottleNet++ achieves interesting compression ratio and bandwidth. Yao et al. [Yao et al., 2020b] proposed a general-purpose offloading framework to reduce end-to-end offloading latency with almost no accuracy loss. By taking the computational capabilities of local and edge devices into consideration, they design an asymmetric encoder-decoder structure that integrates the compressive sensing theory with deep neural networks.

However, all these works have in common that they do not take into account the tradeoff between the size of intermediate data sent to the cloud and the memory size of the layers which are executed at the edge. It is unfortunately an important drawback, as these two aspects are opposite and neglecting one of them could lead to solutions which are not optimal.

The authors of [Hu and Krishnamachari, 2020] propose a streaming inference framework to simultaneously improve throughput and accuracy by communication compression in compact DNNs. They mainly perform three steps in order to optimize the DNN execution: 1) Partition: split the CNN layers to achieve computation load-balance; 2) Compression: insert Auto-Encoders into the original CNN to compress data traffic; 3) Scheduling: adaptively select the compression ratio when the variation of bandwidth is large. Authors say that pipelining input images leads to a higher throughput but without explicit results about the pipelining mechanism cost. In addition to that there are no indications of the performances in case of bulky DNNs as only Mobilenet DNNs were used for experiments.

2.3.3 Neural network optimisation at the edge

Only few works take try to optimise computation at the edge. Matsubara et al. [Matsubara and Levorato, 2020] propose to split DNNs in two parts executed at the mobile device and cloud server respectively. They use this technique for object detection. The core idea is to achieve an in-network compression introducing a bottleneck layer in the early stages of the backbone network. The output of the bottleneck is then sent to the cloud server, which executes some layers to reconstruct the original output. They also use distillation in order to maintain an acceptable performance level. However in this work, only bulky DNN are used and this makes it specific to large hardware resources cameras, and thus not suitable for micro-controllers type architectures for instance. Moreover, the authors only consider Wifi bandwidth scenario.

Jankowski et al. [Jankowski et al., 2020] studied joint device-edge inference considering an IoT device with limited hardware resources (micro-controllers for instance), and a wireless channel to the network edge. They target image classification applications over two criteria: power and bandwidth limited edge device.

This work proposed pruning of the baseline taking into account the noise introduced over the channel, under a constraint on the available bandwidth. They introduce an auto encoder/ decoder on respectively edge and cloud sides in order to achieve good compression rates. This approach seems achieving interesting results in terms of accuracy even with limited computational and communication resources. However this work is only tested on CIFAR-100 dataset whereas it is a small dataset with a limited image resolution (32×32), which cannot accurately represent the benchmark model size or accuracy in real-life case. The only architecture on which this method was applied is VGG-16. It would have been interesting to see whether it can generalise to other types of networks.

2.3.4 Discussion

During last years, there was a multiplication of works proposing to split neural network inference between edge and cloud. This shows that it is a promising solution to overcome the drawbacks of cloud-only and edge-only inference. However, splitting a CNN is also challenging, mainly because the size of the data increases through the different layers. This aspect should be weighed against the memory capacity of the edge : splitting the neural network to far from the input to minimise data size may lead to excessive computation left to the edge, making this solution infeasible for very constrained devices such as MCUs. Hence, there is a need to introduce into these systems compression techniques offering better performance at reducing the size of the transmitted data while minimising the loss of precision and also taking into account the constraints of the edge device.

2.4 Dynamic input-dependent inference in deep neural networks

In most existing partitioning approaches, all input images undergo exactly the same processing in the DNN whereas they often have heterogeneous natures and complexities. This heterogeneous complexities are due to both an inter-class and an intra-class variety. Fig. 2.6 shows an example of the latter. Indeed, even among images belonging to the same class, several factors may impact the complexity of the recognition such as : different poses, presence of occlusions, presence of other objects, lightening issues, scale problems...

Despite this, the same computation is consumed to recognize all instances. All images cross all the DNN layers, whether they are of great complexity or absolute simplicity. This generates additional costs in terms of execution time for simple images for which a decision can be taken well before their arrival at the last stages of the DNN. This is even more critical in constrained cloud-edge environments as resources are very crucial.

Several methods have been proposed in the literature for a dynamic DNN inference adaptive to the input images. Most of these methods target a single execution platform (Section 2.4.1), and other ones target a dual cloud/edge execution platform (Section 2.4.2)

2.4.1 Single execution platform

We find different types of methods to dynamically adapt a DNN inference to each input in the literature [Han et al., 2021]:

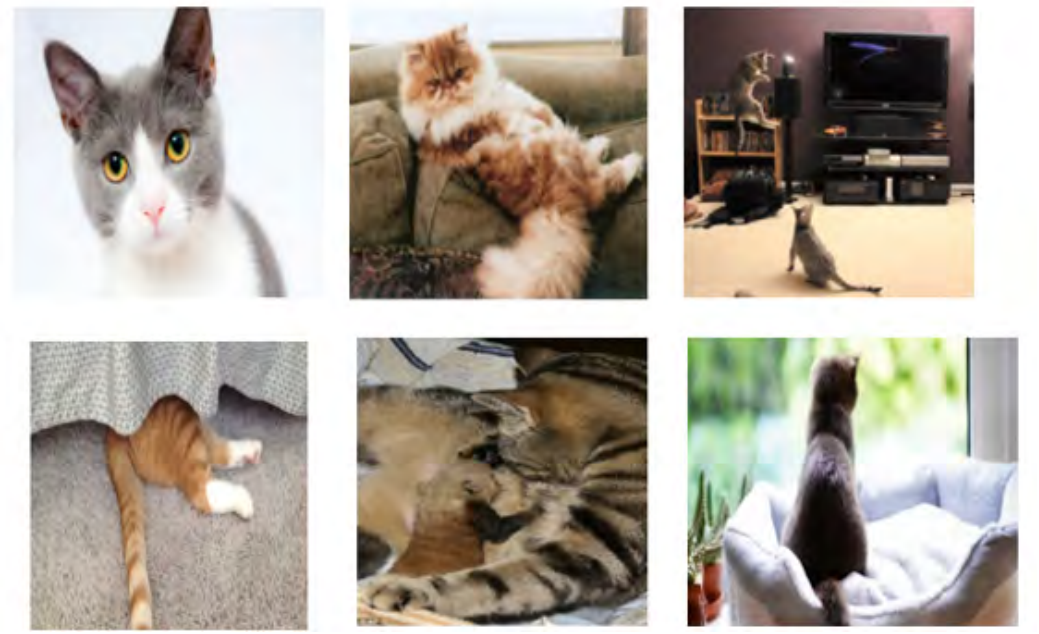


Figure 2.6: Heterogeneous recognition difficulty of test instances

2.4.1.1 Dynamic depth

As modern DNNs are getting increasingly deep for recognizing more ”hard” samples, a straightforward solution to reducing redundant computation is performing inference with dynamic depth, which can be realized by 1) early exiting, i.e. allowing ”easy” samples to be output at shallow exits without executing deeper layers ; or 2) layer skipping, i.e. selectively skipping intermediate layers conditioned on each sample.

1) Early exiting

The most intuitive approach to enabling early exiting is cascading multiple models, and adaptively retrieving the prediction of an early network without activating latter ones. For example, Big/littleNet [Park et al., 2015] cascades two CNNs with different depths. After obtaining the SoftMax output of the first model, early exiting is conducted when the score margin between the two largest elements exceeds

a threshold. The models in this cascading structures are mutually independent. Consequently, once a "difficult" sample is decided to be fed to a latter network, a whole inference procedure needs to be executed from scratch without reusing the already learned features. A more compact design is involving intermediate classifiers within one backbone network, so that early features can be propagated to deep layers if needed.

Berestizshevsky and Even [Berestizshevsky and Even, 2019] studied the trade off between computational effort and classification accuracy in a cascaded DNN. Several intermediate classifiers are included in the DNN. The user sets the acceptable accuracy degradation during inference, which then automatically determines confidence thresholds for the intermediate classifiers. When the confidence threshold is met, the inference process terminates immediately without having to compute the output of the complete network. The authors show that using a softmax output as a confidence measure in a cascade of deep neural networks leads to a reduction of 15%-50% in the number of multiply-accumulate operations operations while degrading the classification accuracy by roughly 1%. However, this approach only optimises accuracy and does not allow to specify time or energy requirements. The authors of BranchyNet [Teerapittayanon et al., 2017b] propose a similar approach with an alternative confidence criteria based on entropy. They learn the early stopping policy considering the entropy of the predicted scores of each classifier.

Huang et al. [Huang et al., 2017a] proposed MSDNet, a network architecture that performs on par with state-of-the-art convolutional networks, whilst facilitating prediction in two settings: (1) an anytime-prediction setting in which the network's prediction for one example is progressively updated, facilitating the output of a prediction at any time; and (2) a batch computational budget setting in which a fixed amount of computation is available to classify a set of examples

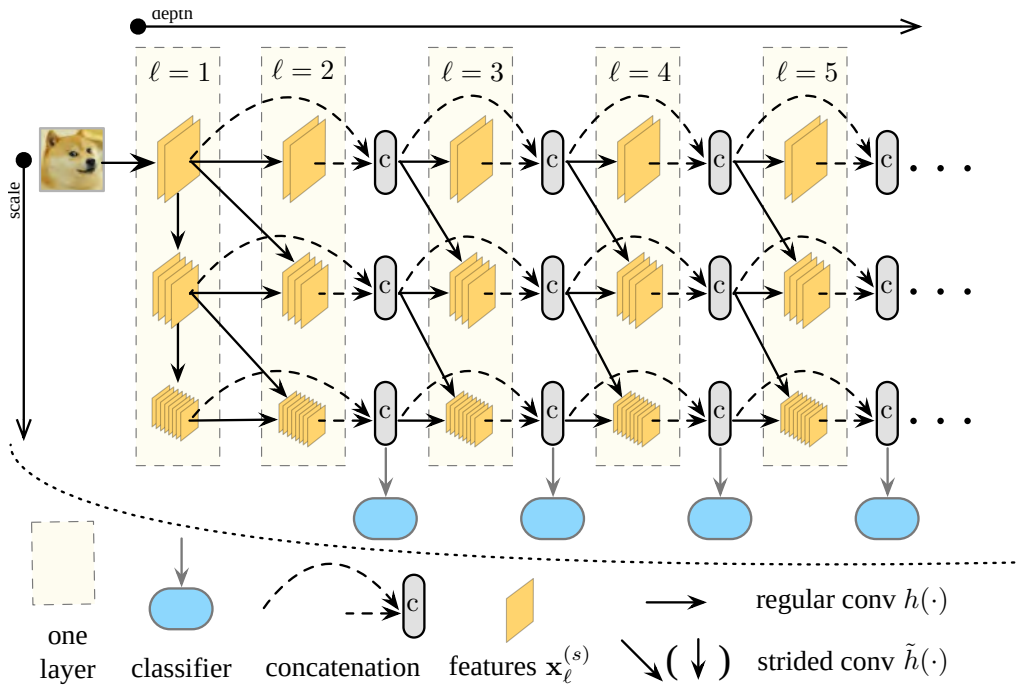


Figure 2.7: Illustration of the first five layers of an MSDNet with three scales. The horizontal direction corresponds to the layer di- rection (depth) of the network. The vertical direction corresponds to the scale of the feature maps. Horizontal arrows indicate a regular convolution operation, whereas diagonal (vertical) arrows indicate a strided convolution operation. Classifiers only operate on feature maps at the coarsest scale. [Huang et al., 2017a]

that can be spent unevenly across "easier" and "harder" examples. This approach uses multi-scale convolutions and progressively growing feature representations, which allows for the training of multiple classifiers at intermediate layers of the network (figure 2.7). The authors introduce multi-scale feature maps and dense connectivity. They show that this approach overpasses traditional early-exit solutions. However, this approach is not designed for edge devices. The additional scales and connections in MSDNet imply computation overhead which is not analysed in the paper.

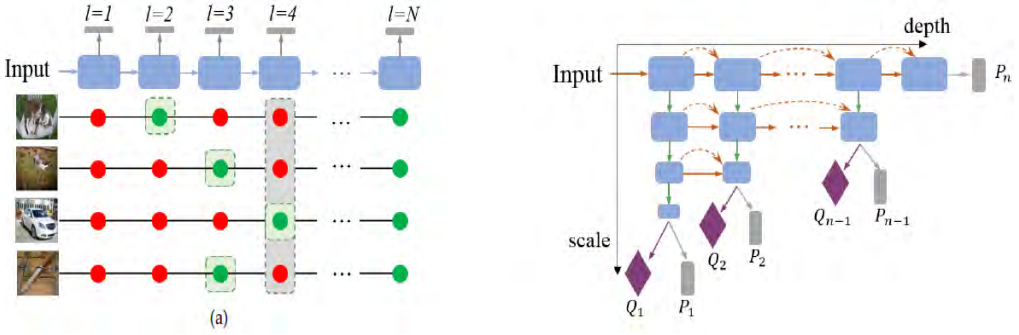


Figure 2.8: Left: for a given test instance, the green and red circles represent the correct and incorrect output predictions respectively. Right : an illustration of the RCN architecture. [Jie et al., 2021]

Jie et al. [Jie et al., 2021] proposed a routing convolutional network (RCN) that has multiple early prediction outputs to classify instances with heterogeneous difficulties, thus reducing the average time cost without losing much accuracy. The authors interestingly observed that the 'latest-all' strategy is not optimal in early exit approaches, meaning that the latest exit within a time budget cannot always produce the more accurate predictions than the earlier exits for a particular sample (figure 2.8). Therefore, the optimal prediction output is selected by training Q-networks that consider the tradeoff between potential time cost and expected information gain. The alternate training mechanism of Q-networks and early prediction outputs has also been proposed to allow the prediction outputs to be better positioned. Experiments on the CIFAR-10, CIFAR-100, ILSVRC 2012, and Cityscape benchmarks validate the superiority of the proposed CNN over the standard CNN with fixed depth and other early stopping methods relying on the predicted scores. However, the execution model overhead (computation resources and memory footprint) make it expensive to scale such approaches up to very deep models. For example, for ResNet-20 (30M parameters), the 8 Q-networks bring 2.5M extra-parameters.

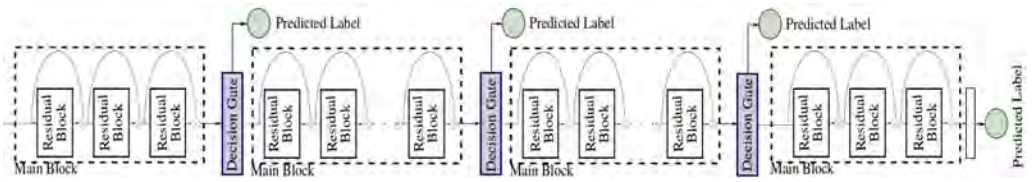


Figure 2.9: Decision gates (d-gate) are integrated directly into deep neural networks, and are trained to predict whether decisions can be made at the d-gate or require projecting into deep embeddings [Shafiee et al., 2019]

Shafiee et al. [Shafiee et al., 2019] introduced a concept of decision gates (d-gate), modules trained to decide whether a sample needs to be projected into a deeper embedding or if an early prediction can be made at the d-gate, thus enabling the computation of dynamic representations at different depths (figure 2.9). The proposed d-gate modules are a kind of general concept that can be integrated with any deep neural network in order to reduce the average computational time while maintaining modeling accuracy. The proposed d-gate framework is examined via different network architectures and datasets.It allows a 55% speedup and 39% FLOPs reduction on DenseNet-201 trained on the CIFAR10 dataset with 2% drop in accuracy. However, each d-gate consists in a small neural network. It can cause an overhead which is not evaluated in the paper, as the authors do not specifically target constrained edge-devices.

Bonato and Bouganis [Bonato and Bouganis, 2021] a method that identifies the best place to attach an early exit, that maximizes the time savings while preserving the original baseline model accuracy, is proposed. Simultaneous multi-layer feature of a given target class are extracted by the mean of the exploration of a dense connection between early exit branch and baseline model layers. The evaluation of the proposed methodology demonstrated an average time savings in the order of 2.09× to 8.79× for the CIFAR10 dataset and 15.00× to 20.71× for CIFAR100 for baseline models ResNet-21, ResNet-110, Inceptionv3-159, and DenseNet-121.

The authors of [Wołczyk et al., 2021] introduce Zero Time Waste (ZTW), an approach in which early exits have an Internal Classifier (IC) that reuses predictions returned by its predecessors by adding direct connections between ICs and combining previous outputs in an ensemble-like manner. These classifiers are introduced on each early exit, adding a significant complexity to the execution devices.

2) Layer skipping

Wu et al. [Wu et al., 2018] introduce BlockDrop, an approach that learns to dynamically choose which layers of a DNN to execute during inference so as to best reduce total computation without decreasing prediction accuracy. This method exploits the robustness of Residual Networks to select on-the-fly which residual blocks to evaluate for a given novel image. Thus, rather than executing all blocks of a ResNet, the proposed approach learns a policy to select the minimal configuration of blocks that is needed to correctly classify a given input image (figure 2.10). Specifically, each input sample is first processed by the policy network, whose output directly determines which parts of the main network should be activated. One limitation of this scheme is that the architecture and the training process of the policy network is developed for a specific backbone, and cannot be easily adapted to different architectures.

Skipnet [Wang et al., 2018] enables skipping layers dynamically during inference using small gating networks. The gating networks make binary decisions on whether skipping the calculation of a residual block. The whole network is trained in two phases; first relaxing some parameters for the gating network then a dedicated algorithm is applied to refine the gating network separately. The gating functions are implemented as a Recurrent Neural Network (RNN). An extension of this work is presented in [Shen et al., 2020], where a layer can be partially

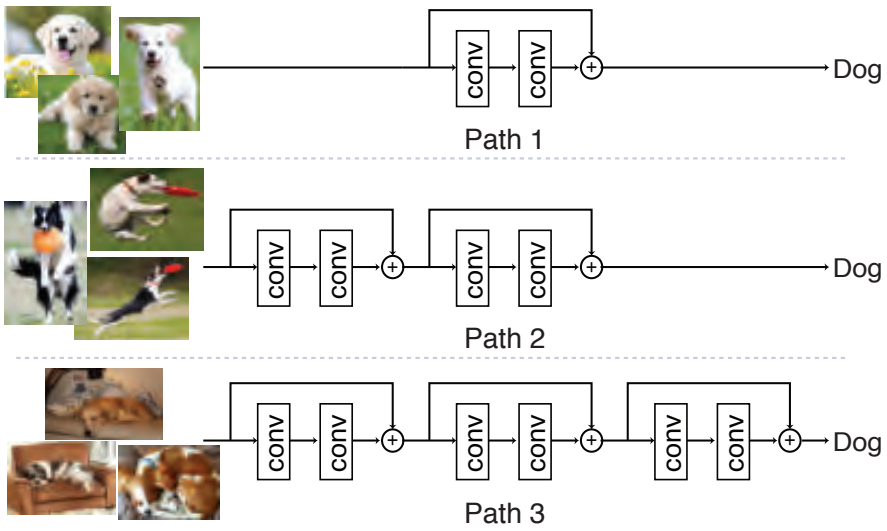


Figure 2.10: A conceptual overview [Wu et al., 2018]

skipped as well, outperforming the previous SkipNet in time saving and prediction accuracy.

2.4.1.2 Dynamic width

In addition to dynamic network depth, a finergrained form of conditional computation is performing inference with dynamic width : although every layer is executed, its multiple units (e.g. neurons, channels or branches) are selectively activated conditioned on the input.

Bengio et al. [Bengio et al., 2016] propose an approach allowing to skip some neurones in Fully connected (FC) layers. This approach is based on the idea that different neuron units are responsible for representing different features, and therefore not all of them need to be activated for every sample. The authors cast the problem of learning activation-dependent dropout policies for blocks of units as a reinforcement learning problem. Prior to this study, other works proposed

similar neurons conditional computation approaches [Bengio et al., 2013,Cho and Bengio, 2014].

Modern CNNs usually have considerable channel redundancy. Based on the common belief that the same feature channel can be of disparate importance for different samples, adaptive width of CNNs could be realized by dynamically activating convolutional channels. The authors of [Lin et al., 2017] propose an approach to decide the execution of channels at every layer by gating functions. They introduce runtime neural pruning (RNP), which models the layer-wise pruning as a Markov decision process, and a RNN is used to select specific channel groups at every layer. More recently, Slimmable neural network [Yu et al., 2018] uses a gate module to decide the width for a whole stage of a ResNet. The model complexity is dynamically adapted by exploiting the width of the feature map. A switchable batch normalisation is introduced to deal with statistics among different model variants (switch).

Rather than placing plug-in gating modules inside a CNN, GaterNet [Chen et al., 2018] builds an extra network, which takes in the input sample and directly generates all the channel selection decisions for the backbone CNN. This implementation is similar to [Wu et al., 2018] that exploits an additional policy network for dynamic layer skipping.

Even if this approaches enable to save computation at inference time, they don't allow early predictions, which can be crucial in a distributed setting to get local inference at the edge.

2.4.1.3 Dynamic routing

The aforementioned methods mostly adjust the depth or width of classic architectures by activating their computational units conditioned on the input. Another

line of work develops different forms of SuperNets with various possible inference paths, and performs dynamic routing inside the SuperNets to adapt the computational graph to each sample.

Liu et al. [Liu and Deng, 2017] introduces a new type of feed-forward deep neural network that allows selective execution. Given an input, only a subset of neurons are executed, and the particular subset is determined by the DNN itself. By pruning unnecessary computation depending on input, the DNN provide a way to improve computational efficiency. To achieve dynamic selective execution, this approach uses controller modules. Each controller module is a sub-network whose output is a decision that controls whether other modules can execute. Both regular and controller modules in a DNN are learnable and are jointly trained, by integrating back propagation with reinforcement learning, to optimize both accuracy and efficiency. However, this type of approach where additional sub-networks must be run to take decisions during inference is not applicable when constrained edge devices are implied.

As decision trees always perform inference along one forward path that is dependent on input properties, combining tree structure with neural networks can naturally enjoy the adaptive inference paradigm and the representation power of DNNs simultaneously. Soft decision tree (SDT) [Rota Bulo and Kontschieder, 2014, Frosst and Hinton, 2017] adopts neural units as routing nodes (blue nodes in Fig. 5 (c)), which decides the portion that the inputs are assigned to their left/right sub-tree. Each leaf node generates a probability distribution over the output space, and the final prediction is the expectation of the results from all leaf nodes. Although the probability for a sample reaching each leaf node in an SDT is data-dependent, all the paths are still executed, which limits the inference efficiency.

2.4.2 Dual execution platform

More recently, some works have explored dynamic inference for distributed execution platforms. All these approaches consider a single client at the edge level.

Edgent [Li et al., 2018] is a collaborative and on-demand DNN co-inference framework with device-edge synergy. It trains models with multiple exit layers, mainly in the later stages. Based on observed bandwidth, they greedily search the best model partition point and exit point. Authors demonstrate effectiveness of their approach in terms of latency but unfortunately did not provided data regarding accuracy.

Yao et al. [Yao et al., 2020a] present a real-time scheduling algorithm for intelligent real-time edge services for voice recognition, LIDAR processing, or machine vision, on behalf of local embedded devices that are themselves unable to support extensive computations. The work proposes a scheduling algorithm and its goal is to maximize average accuracy of deep neural network outputs, while meeting task deadlines, thanks to opportunistic shedding of the least necessary optional parts. The dynamic and generic aspects of this approach seems interesting, however the scheduling overhead might be important, especially as no efficient compression is used in order to minimise edge outputs.

The authors [Zeng et al., 2019] propose Boomerang, an on-demand cooperative inference framework under the IoT environment. It introduces DNN right-sizing and DNN partition to DNN models, where DNN right-sizing reshapes the computation of the DNN model in the early-exit mechanism, and DNN partition enables cooperative inference execution between the IoT device and the edge server. This approach looks at optimizing only the bandwidth without the possibility to introduce any other criteria.

In [Laskaridis et al., 2020b] the authors propose SPINN, a distributed inference system that employs synergistic device-cloud computation together with a progressive inference method to obtain fast CNN inference. The proposed system introduces a scheduler that optimizes the early-exit policy and CNN splitting at run time. This approach is able to deliver higher performance over the state-of-the-art systems across diverse settings, without sacrificing the overall system's accuracy and availability.

2.4.3 Discussion

During the last layers, many different solutions were proposed to allow a dynamic input-dependent inference for deep neural networks. These approaches enable to save computation and time by allocating less computation efforts to 'easy' instances. However, most of them target a single execution platform, mostly the cloud, without taking into account the constraints implied by edge execution or hybrid cloud/edge execution, such as the network bandwith, the size of the data which is sent to the cloud, and limited memory/power of edge devices. As a result, most of them are not easily transposable to a cloud/edge execution environment. The approach which seems to be the easier to adapt to this hybrid execution scheme is the early-exit solution. Recent works have shown that putting an early-exit in the edge node enables to get local predictions, thus saving bandwidth. However, all these approaches target a cloud/single-edge environment, without analysing the possibility to extend it to a multi-edge distribution context. Morever, none of them allows to optimise time, accuracy and energy in a same solution.

2.5 Cloud/Multi-edge DNN partitioning

Even if they are few in number, some recent works try to extend DNN approaches for image / video processing to the multi-camera context.

The authors of [Xu et al., 2020b] propose a task model offloading algorithm by considering how to optimally deploy neural network model into the edge nodes (figure 2.11). They provide an adaptive task scheduling algorithm to control and manage the task assignment by using the improved ant colony algorithm. Each task is a single DNN layer for a specific input : each inference is divided blindly in several independent tasks for each layer and may be affected to different edge nodes. Knowing the data amplification problem (Section 2.3.1), this method implies important communication costs and may lead to sub-optimal solutions. This is accentuated by the fact the the authors don't propose a solution to compress intermediate data.

The authors of [Teerapittayanon et al., 2017a] propose a novel distributed deep neural network architecture (DDNN) that is distributed across computing hierarchies, consisting of the cloud, the edge and end devices (cameras). All cameras execute the same portion of the DNN, with a single splitting point. They don't factor in the significant communication overhead introduced by the size of intermediate layers outputs, and energy is not considered.

The core problem addressed in [Price et al., 2018] is how to achieve on-board, real-time, continuous and accurate vision-based detections using DNNs for visual person tracking through micro aerial vehicles (MAV). The proposed approach leverages cooperation among multiple MAVs. First, each MAV fuses its own detections with those obtained by other MAVs to participate in a complex cooperative visual tracking. This allows for predicting future poses of the tracked person,

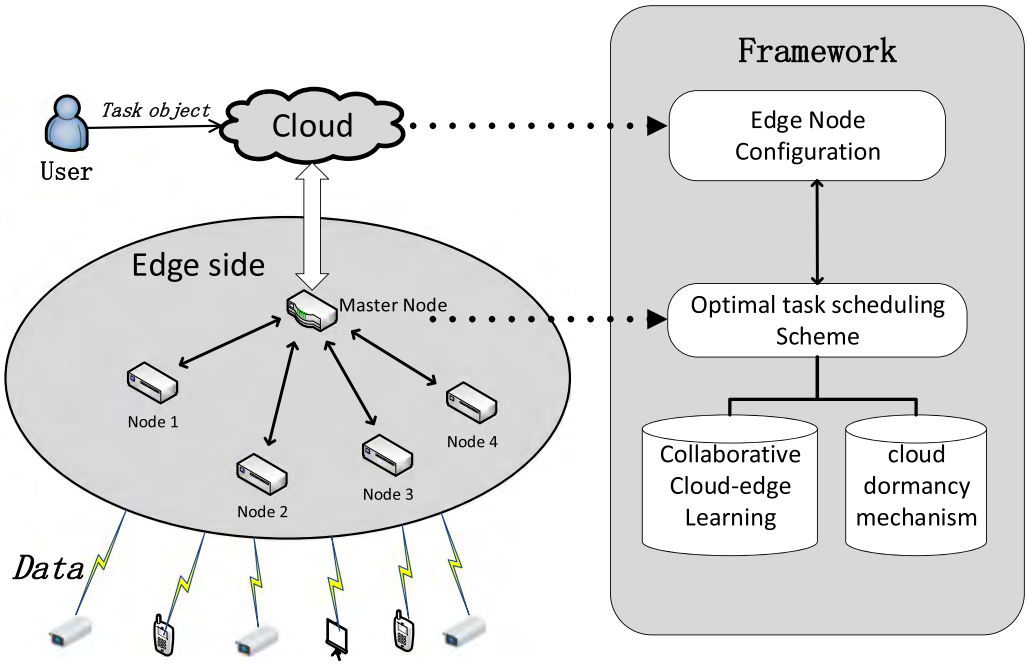


Figure 2.11: The collaborative cloud-edge computing framework [Wu et al., 2018]

which are used to selectively process only the relevant regions of future images, even at high resolutions. This solution targets a hardware platform composed by CPUs and GPUs on distributed nodes without any connection to the cloud. Thus all computations are processed on local hardware. The idea of distributed control is interesting, but this approach supposes using cameras with powerful computation elements (GPUs) able to run a whole DNN. Thus it is not adapted to small size cameras with micro controllers type hardware.

Zhao et al. [Zhao et al., 2018] partition a convolution layer into tiles that are distributed among edge devices. Its scheduler performs work-stealing for runtime workload-balance. The authors work propose a framework for adaptively distributed execution of CNN-based inference applications on tightly resource-constrained IoT edge clusters. DeepThings employs a scalable Fused Tile Partitioning of convolutional layers to minimize memory footprint. It includes a dy-

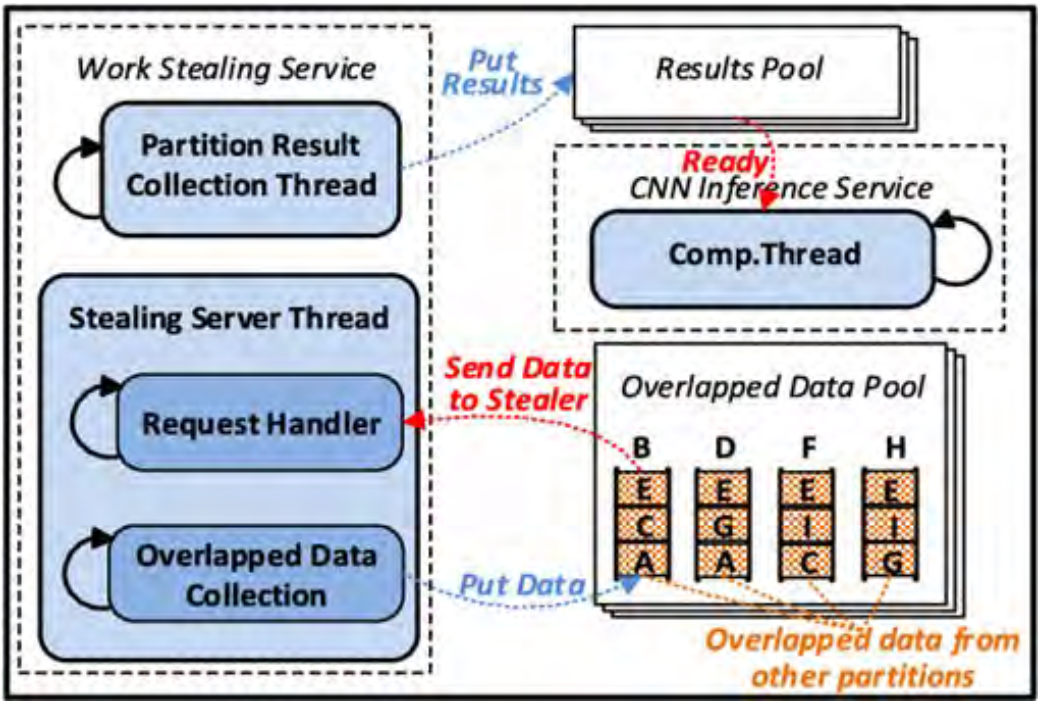


Figure 2.12: Reuse data management in [Zhao et al., 2018]

namic workload distribution and balancing approach at inference runtime. This method integrates a work scheduling process to improve data reuse as shown in fig. 2.12. Results show that this method improves significantly memory footprint without sacrificing accuracy. The problem with such an approach is the scheduler complexity, which seems not included in terms processing time and memory footprint results.

2.6 Summary

In this section, we have presented existing works related to edge computing, dynamic input-dependent DNN inference, and distributed cloud-edge DNN inference. These research areas have been very active during the last few years due to the expansion of DNNs and the development of end-devices technologies. The

distribution of a DNN between multiple devices and the cloud remains an important research that have been addressed by few works. None of them covers all the aspects related to such an hybrid execution, in particular the limited edge-devices' memory and power, as well as the important size of intermediate layers outputs.

Throughout the thesis it is assumed that the aspects linked to network transactions security and privacy are ensured by an upstream mechanism. The security and privacy of data during transmission is therefore not covered here. Indeed, privacy is a key motivation for using edge computing. In edge/cloud computing, the fact that intermediate data is transmitted to the cloud instead of raw data maintains a level or privacy. It is of course possible to reconstruct the original data from intermediate data, but it adds a level of difficulty. The reconstruction is impossible if upstream mechanisms, such as encryption, are used. Moreover, the cloud/edge solution has the advantage of preserving the privacy of the DNN model, which is not completely shared with the edge.

In the next chapter, we propose a technique to partition the DNN inference between the cloud and a single edge device. The objective is to propose a solution to the trade off between the amount of memory at the edge and the size of intermediate data. Indeed, as seen in this chapter, this aspect has never been studied in the literature.

3 | Cloud/Single-edge DNN partitioning

In Chapter 2, we showed that there is a need to introduce edge/cloud solutions reducing the size of the transmitted data while taking into account the memory constraints of the edge device. This is the objective of this chapter, where a novel technique to partition DNN inference between the cloud and a single edge is presented.

This chapter is organised as follows. First, we introduce the proposed approach (Section 3.1), before presenting the architecture design (Section 3.2) and the learning methods (Section 3.3). Then, we present our experiments, including implementation details (Section 3.4), a sensitivity analysis (Section 3.5), and a comparison with related works (Section 3.6), before concluding (Section 3.7).

3.1 Introduction

As explained is Chapter 2, an alternative option to cloud-only and edge-only computing is to distribute the inference of machine learning algorithms between the edge and the cloud.

Towards this goal, several key aspects must be taken into account. The first one is the memory capacity of the edge device : the model must fit the memory of the edge device, which is particularly challenging with very limited edge-devices (such as MCUs). A second important aspect is the size of the intermediate data : this is related to the data amplification issue, as explained in Section 2.2.1. This problem refers to the fact that the output data of the intermediate layers of a DNN are typically much larger than the original image. As seen in Section 2.2, no existing work proposes to jointly optimise these two aspects.

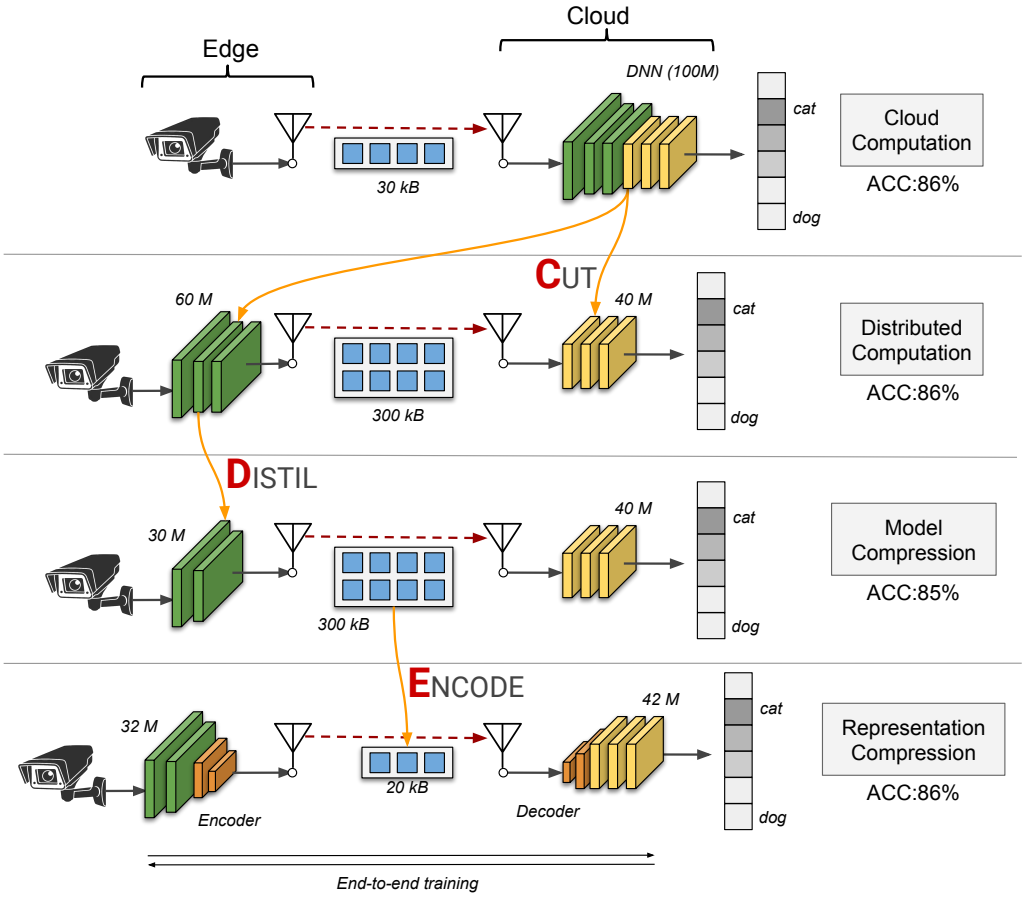


Figure 3.1: The overall process of Cut, Distil and Encode (CDE). In the **Cut** phase, the network is partitioned at a particular layer. In the **Distil** step, the edge-based sub-network is optimized using knowledge distillation, reducing memory and computational load on the edge. Lastly, in the **Encode** step, the intermediate network data is compressed, reducing bandwidth.

Hence, there is a need to introduce into these systems compression techniques offering better performance at reducing the size of the transmitted data while minimising the loss of precision and also taking into account the constraints of the edge device. In this chapter, we propose a novel distributed DNN which distributes the network inference between the edge and the cloud. The key to our approach is a fusion between an auto-encoder and knowledge distillation which

enables end-to-end training of the network while respecting the computational resources available on the edge and the bandwidth provided by the networking protocols (as described in Fig.3.1). In particular, knowledge distillation is utilised to compress the subset of the networks that run on the edge such that real time inference can be achieved while at the same time maintaining the overall neural network's accuracy by transferring the knowledge from the original network (teacher) to the distilled network (student). More precisely, the novel contributions are the following:

- We propose the first neural network partitioning approach considering both resource constraints at the device and bandwidth constraints. We are jointly optimising memory (parameters on the edge through distillation) and communication (size of intermediate representation through encoding).
- Towards this goal, we introduce a novel model by jointly training knowledge distillation at the edge and an auto-encoder. The proposed method enables us to partition the neural network execution with no accuracy drop while respecting important memory and bandwidth constraints.
- We provide an unexpected insight regarding the behaviour of the network: our results show that it is not always beneficial to push as much computation as possible on the device. Under some conditions, it may be better to split the network at lower layers for a better accuracy with an identical communication cost, even if the device can afford more computation.

Note that in this Chapter, 'low' layer refers to earlier layer in the DNN architecture, whereas 'high' layer refers to later layer in the DNN architecture.

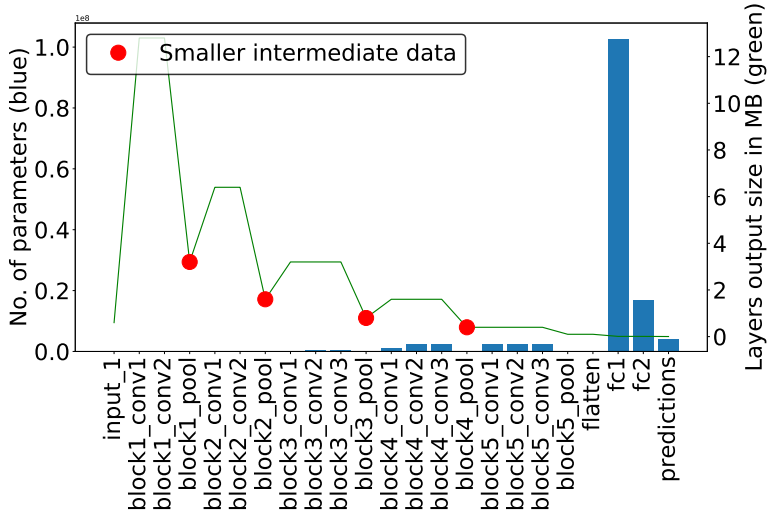
3.2 Architecture design

3.2.1 Inflection points

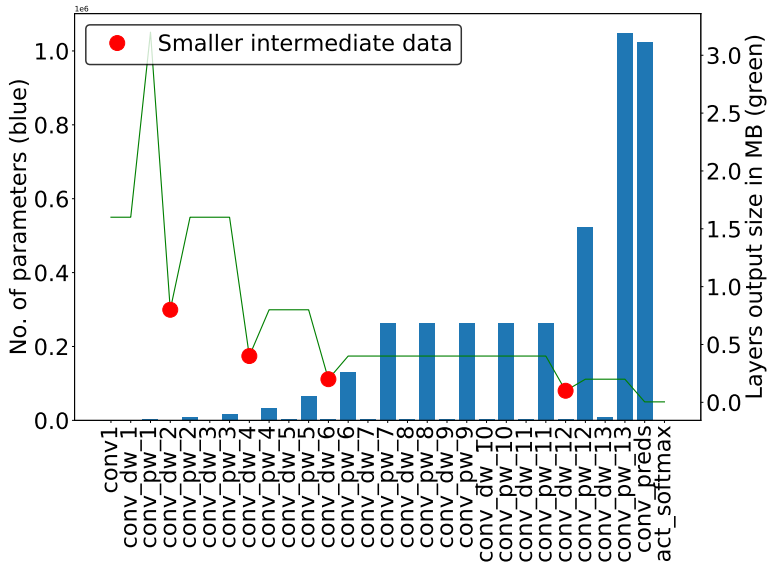
Dividing the inference of the DNN between the node and the cloud at the most fundamental requires careful choice of the layer at which to split, which must satisfy certain constraints. It must represent the best compromise regarding memory, bandwidth and precision. As shown in Fig. 3.2 (a) and Fig. 3.2 (b), the curves representing the output data sizes of each layer, whether for MobileNetV1 or VGG16 networks, have easily recognizable inflection points. These points correspond to the layers for which the output data sizes are local minima of the curves. This observation restricts the dimensionality of the search space and helps choosing the suitable layers for network splitting.

However, as illustrated in Table 3.1, even when choosing these layers to cut the networks, the output size of the intermediate data is too high for real-time applications with limited bandwidth such as LoRa (4.68KB/s) or even 3G (230KB/s). Moreover, as explained in Section 1 in Table 2.2, the size of the first part of network is still big for edge devices with memory capacity limited to few KB or MB.

Therefore, we designed a framework aiming at optimising these different constraints: memory size at the edge, size of the transmitted intermediate data, and the model accuracy. We focus this study on the challenging trade off between memory and bandwidth.



(a) VGG16



(b) MobileNetV1

Figure 3.2: The size of output intermediate data and number of parameters of each layer for (a) VGG16 and (b) MobileNetV1. Inflection points (potential splitting points) are marked in red.

Table 3.1: The size of intermediate data when splitting MobileNetV1 and VGG16 at different layers.

Model	Layer	Size of intermediate data
MobileNetV1	28	392KB
MobileNetV1	41	191.5KB
VGG16	7	2MB
VGG16	11	784KB
VGG16	15	392KB

3.2.2 Architecture description

The proposed architecture is composed of three main parts (as seen in Fig.3.3) which are motivated by the constraints presented in the previous section, and which are trained end-to-end as explained in the next section.

The first part is the part which is executed on the edge device. This part is compressed using the knowledge distillation technique (KD) in order to fit the memory constraints of the device.

The second part, at the center of the architecture, is an auto-encoder (Auto encoder (AE)) aiming at reducing the size of the intermediate data, like a bottleneck. This is an under complete convolutional auto-encoder using ReLU as activation function after each layer. The first block of the auto-encoder consists of layers whose size gradually decreases (encoding). It is followed by a second block of layers, which are increasing the size of the data to match the size of the second part of the network (decoding). Therefore, the size of the intermediate data which is sent across the network is determined by the size of the auto-encoder's code. The execution of the auto-encoder is split between the edge and the cloud.

The third part simply is the remainder of the original network, which is executed on the cloud.

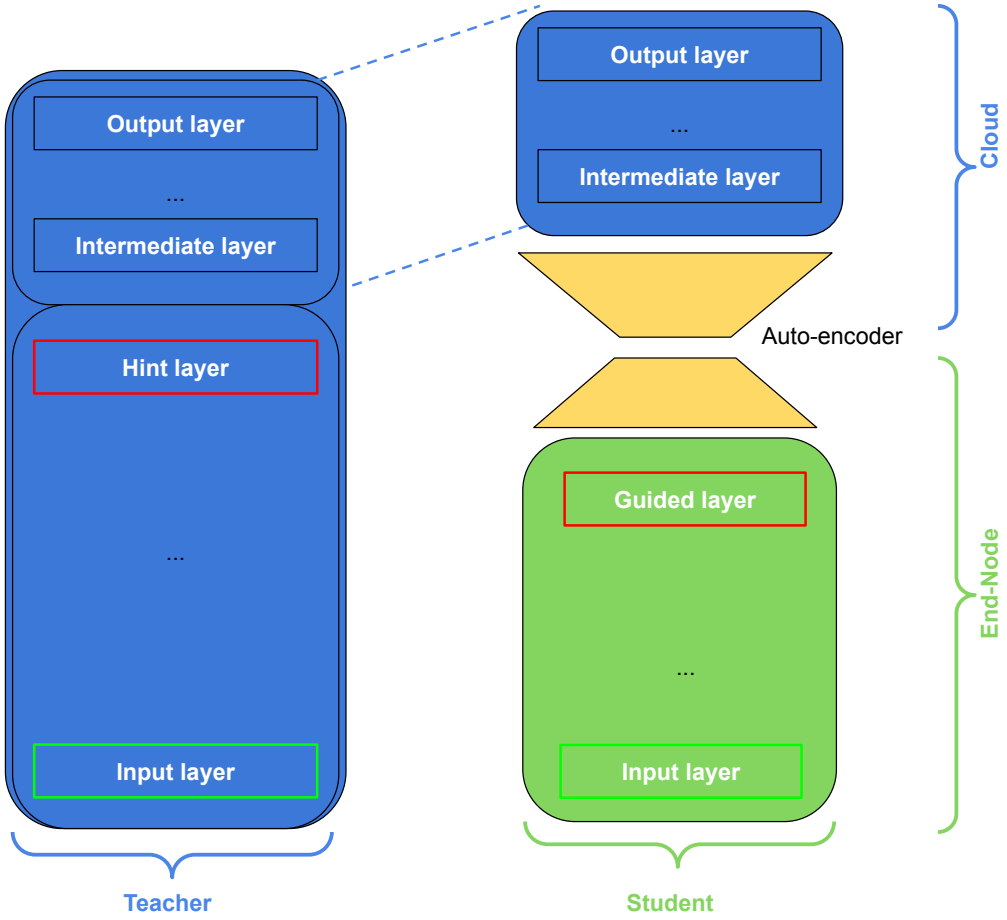


Figure 3.3: The distributed DNN architecture.

Hence, we propose an architecture based on the distillation of the part executed on the edge only, and preserving the cloud part without distillation. Indeed, as resource constraints (memory, computation, energy) being much more important at the edge than in the cloud, it is not useful to distill the whole network and incur a decrease in accuracy.

We can expect several advantages from this approach. Compressing part of the network reduces the local memory consumption considerably while the auto-encoder reduces the amount of data transmitted to the cloud. The end-to-end

training described in the previous section aims at maintaining an acceptable accuracy rate (which might even improve in some cases) by jointly training these two optimisations.

Furthermore, the distillation technique is generalisable and applicable on several types of networks: MobileNet, VGGNet, ResNet, etc., without requiring bespoke edge architectures to be created.

Finally, the distillation technique and the auto-encoder are flexible, on the same network, in depth (number of layers) and in width (number of filters), which allows adaptation with respect to the bandwidth and memory capacity at the edge.

3.3 Learning methods

3.3.1 Training the student network at the edge

In order to train the whole model, we firstly need to train the small student network S such that the output of the intermediate layer of S is similar to the output of the intermediate layer of the teacher network T. We achieve this by employing a Hint training (HT) procedure [Romero et al., 2015, Saputra et al., 2019] such that S can mimic the T's capability to generate useful latent representation for the task. In HT, a *hint* is defined as a layer in T that is utilized to guide a *guided* layer in S. In other words, the guided layer is trained to reproduce the same output as the hint layer. The objective function is the mean-squared error between the output of these two layers. More specifically, let $\mathbf{W}_{hint}$ and $\mathbf{W}_{guided}$ be the weights of T and S up to their hint and guided layers respectively. Then, given $\{\mathbf{I}_i, i = 1, 2, ..., n\}$ as the n number of training data, the network will be trained by using the following

objective function

$$\mathcal{L}_{KD} = \frac{1}{n}\sum_{i=1}^{n} \left\| \Phi_T(\mathbf{I}_i; \mathbf{W}_{hint}) - \Phi_S(\mathbf{I}_i; \mathbf{W}_{guided}) \right\|^2, \tag{3.1}$$

where Φ_T and Φ_S are T's and S's DNN functions up to their respective hint or guided layers. Note that the splitting points of Φ will be different between T and S.

3.3.2 Auto-encoding intermediate data

After training the student network in the edge device, we compressed S's latent representation by using Auto-Encoder (AE) in order to respect the desired communication requirements (e.g., 3G/LoRa). In that sense, AE will be trained to generate a compressed S's representation while at the same time should be able to reproduce the original S's representation to be consumed by the part of the network in the cloud device. Let $\mathbf{W}_{AE}$ be the weight of the AE, then in the second stage, the network is trained by using this loss function

$$\mathcal{L}_{AE} = \frac{1}{n}\sum_{i=1}^{n} \left\| \Phi_S(\mathbf{I}_i) - \Psi_S(\Phi_S(\mathbf{I}_i); \mathbf{W}_{AE}) \right\|^2, \tag{3.2}$$

where Ψ_S is the AE part of S. The example of the student's and the auto encoder's output latent representations after training can be seen in Fig.3.4.

3.3.3 Fine-tuning

Finally, to improve the model performance, we can optionally fine tune the whole network in an end-to-end manner by minimizing the task specific objective func-

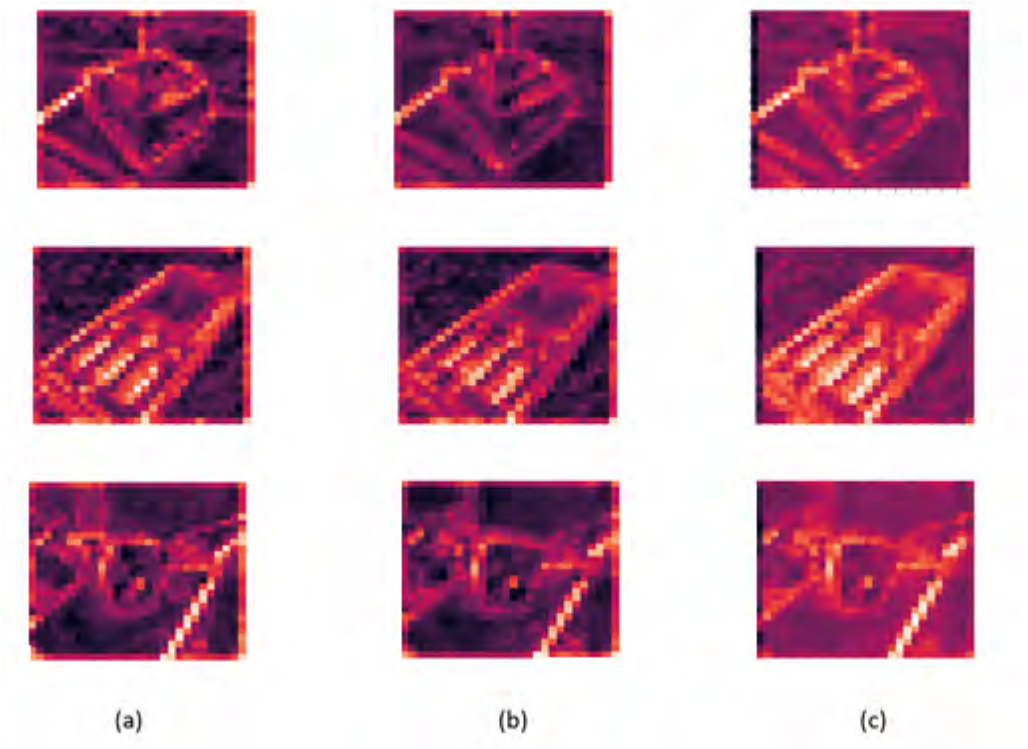


Figure 3.4: Visualisation of the output latent representation from: (a) the hint layer, (b) the guided layer, and (c) the auto-encoder's output. Note that our auto-encoder can reconstruct the hint/guided layer's representation accurately.

tion (e.g., classification, regression) using the following objective function

$$\mathcal{L}_{task} = \frac{1}{n}\sum_{i=1}^{n} \left\| \Omega_S(\mathbf{I}_i; \mathbf{W}_S) - \hat{y}_i \right\|^2 , \tag{3.3}$$

where Ω_S, $\mathbf{W}_S$, and $\hat{y}$ are the full S's DNN layers, the full weight of S, and the true label respectively.

3.3.4 Training process

Regarding the implementation, we performed the training process on the cloud and copied the student and the encoder models to the edge. About the different settings we used during training (e.g., splitting layer), we chose as hint layers the

layers which are optimal in terms of size of intermediate data as illustrated in Fig.2 (e.g., layers 28 and 41 for MobileNetV1, and layers 7, 11 and 15 for VGG16). The training details are described in next section.

3.4 Implementation details

We implemented our model using Keras framework and the model was trained and tested on GPU from Google Colab. For all experiments, we used a subset of ImageNet dataset (700 classes) [Sylvain Gugger, 2023] with medium image definition (224x224). To demonstrate the effectiveness of our methods, we employed VGG16 [Simonyan and Zisserman, 2014] and MobileNetV1 [Howard et al., 2017] with the baseline accuracy 70% and 86.5% respectively in the used dataset. Although MobileNet can run locally on a smartphone, it is still important to optmise it in the context of multiple applications running simultaneously. Moreover, optimising MobileNet makes it possible to use it at a device with more stringent constraints, such as MCUs.

In the first stage of training, we trained the student network (the lower part of the network deployed in the edge) with Adam optimizer [Kingma and Ba, 2014] for up to 500 epochs. We used 10^{-4} as learning rate.

In the second stage of training, we trained the auto-encoder to compress the output of the student network by using Adam optimizer with learning rate 0.005 for up to 500 epochs.

Finally, we bring together the student, the auto-encoder, and the upper-part of the teacher, and jointly trained them using Adam optimizer with $1e-6$ learning rate for up to 100 epochs.

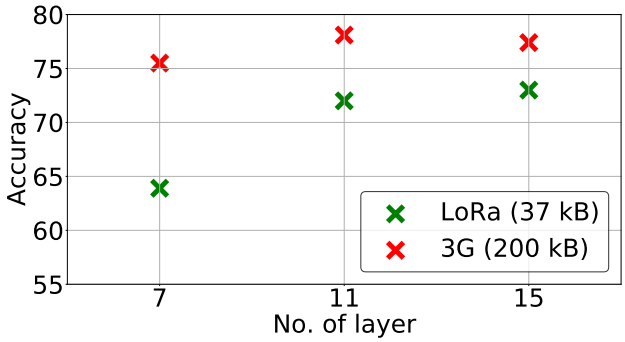


Figure 3.5: Accuracy of VGG16 when splitting the network at layers 7, 11 and 15 for both LoRa (37 KB) and 3G (200KB) communication constraint.

For all stages, we divided 20% of the training data as the validation set and used 32 as the batch size. We performed more than 270 end-to-end trainings with different architectures and training parameters. We present the more significant results in the next section.

3.5 Sensitivity analysis

3.5.1 The Impact of Cutting at Different Layers

In this experiment, we study the impact of accuracy when cutting the network at different layers. Given the communication constraint for 3G (200KB) and LoRa (25KB), we want to see at which layer the best performance is obtained. Note that we only cut a layer that has a better chance to respect the communication constraints as we described in Fig.3.2. The results for this experiment are illustrated in Fig. 3.5 and Fig. 3.6.

While previous works [Emmons et al., 2019] suggest that it is better to split the network as far as possible from the input (because the intermediate data is smaller), especially when we do not have any constraint on the edge capacity,

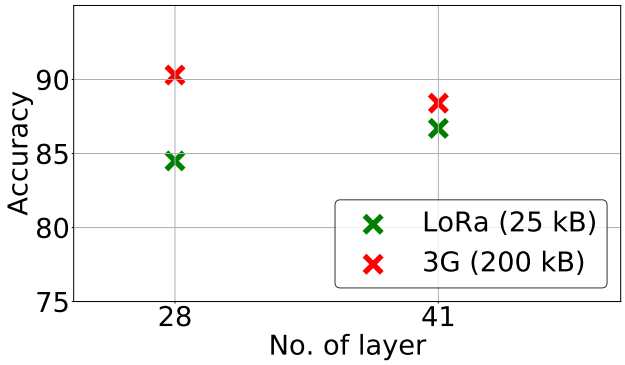


Figure 3.6: Accuracy of MobileNetV1 when splitting the network at layers 28 and 41 for both LoRa (25 KB) and 3G (200 KB) communication constraint.

the result from Fig. 3.5 and Fig. 3.6 indicates that this depends on the bandwidth constraint.

For a LoRa constraint (25KB), VGG16 and MobileNetV1 produces the best performance when we cut the network at layer 15 and 41 respectively, whereas for a 3G constraint (200KB), better results are obtained when splitting these networks at layers 11 and 28 respectively, not the upper ones. These results are quite unexpected, because we could think that as the size of the intermediate data decreases, we can always obtain a better accuracy when cutting the network at higher layers. These results could be explained by the fact that knowledge distillation produces optimal results when the hint layer is chosen in the middle layers.

Another interesting remark from this experiment is that even if we use LoRa bandwidth (25KB) as the communication constraint, MobileNetV1 still produces accurate classification, very close to those obtained with the 3G communication constraint. This is because the MobileNetV1 model contains many normalisation layers which could potentially improve the generalization capability even when we introduce a bottleneck layer.

3.5.2 Trade-off Between Communication Cost and Accuracy

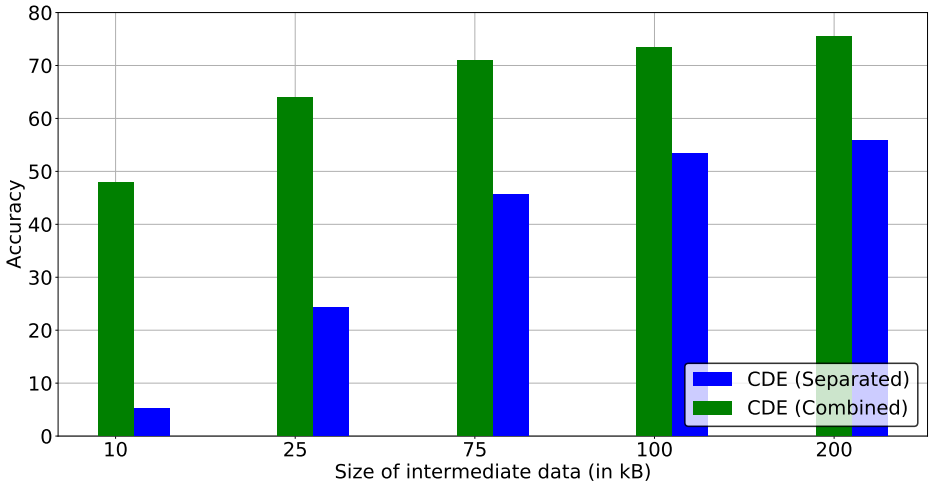
In this section, we measure the impact of communication cost on accuracy when we split the network at a given layer. We use 3G bandwidth capacity, i.e. 200KB, as the limit. We also perform an ablation study by comparing joint training CDE (combined) and separated training CDE (separated). Fig.3.7 show the result.

In all cases, CDE (combined) yields more accurate classification compared to CDE (separated). In general, the model and data compression performed by distillation (D) and auto-encoder (E) might cause a loss of information. By training the network jointly via CDE (combined) approach, the upper part of the network (which resides on the cloud) can be refined, adapting with the new (encoded) feature information.

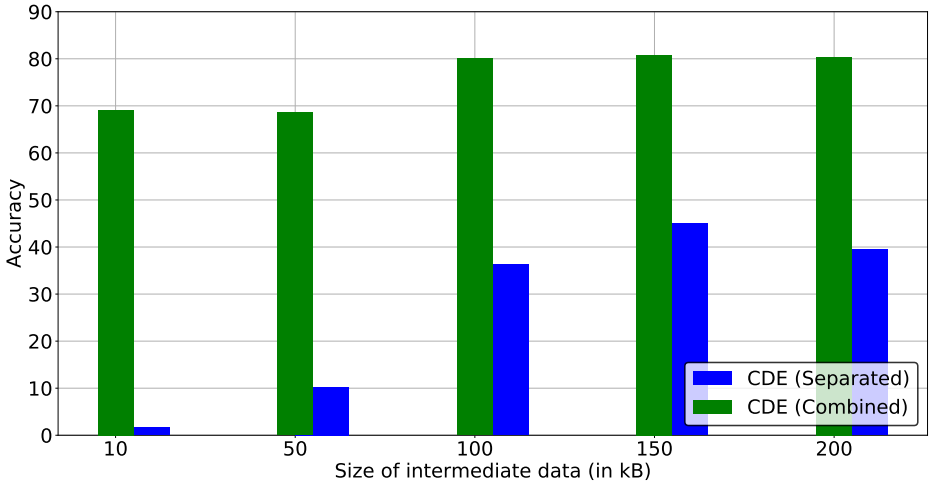
Another observation that we can see from Fig.3.7 is, in general the smaller the encoded data is, the lower is the accuracy. As we have hypothesised, with the smaller size of encoded data, we lose more fine-grained feature information such that accurate classification is hard to achieve.

In the low bandwidth scenario, the difficulty to generate accurate classification is even more pronounced in VGG16 than in MobileNetV1. For VGG16, in order to respect the constraint for 3G communication (i.e. 200KB), we have to compress the intermediate data from 2MB to 200KB (10% of than the original data). On the other hand, we only require to perform 50% compression for MobileNetV1 to satisfy the 3G communication constraint (as can be seen from Table 3.1). In that case, a much higher reduction in accuracy is expected for VGG16.

Nevertheless, our approach is not only able to retain the accuracy of VGG16 with 200KB encoded data, but even surpasses the performance of the original network with 75.5% classification accuracy.



(a) VGG16



(b) MobileNetV1

Figure 3.7: The impact of communication cost on accuracy when we split (a) VGG16 at layer 7 and (b) MobileNetV1 at layer 28, with separated and combined training of KD + AE, with different sizes of intermediate data in KB.

3.5.3 Trade-off Between Memory Size and Accuracy

In this section, we employed both the bandwidth and the memory constraint on the edge device, and measured the impact on accuracy. Note that the memory size of the edge device was ignored in the previous experiments. Given the bandwidth

constraint, we constructed several student networks (part of the whole network that resides on the edge device) to generate models with different memory sizes. The same layer cutting point can generate different network sizes because the students have different architectures depending on the constraint on memory size: a bigger student has more convolution layers and/or more neurons in each layer. The results are presented in Table 3.2 (3G) and Table 3.3 (LoRa).

Table 3.2: The accuracy of VGG16 and MobileNetV1 for different memory sizes at the edge with 3G (200 KB) communication constraint. The white rows indicate that the model size can meet the MCUs capacity. We also indicate the accuracy and memory of the unmodified approach (Teacher T). This is the original accuracy without compression of the raw data.

	Splitting layer	Memory size at the edge	Accuracy
	No Splitting (T)	553.16 MB	70%
VGG16	Layer 7	320.3 KB	70.9%
	Layer 7	324.4 KB	73.4%
	Layer 7	340.8 KB	75.5%
	Layer 11	1.0 MB	72.2%
	Layer 11	1.1 MB	78.1%
	Layer 15	3.9 MB	70.4%
	Layer 15	4.2 MB	77.4%
	No splitting (T)	17.3 MB	86.5%
MobileNetV1	Layer 28	161.8 KB	80.8%
	Layer 28	244.3 KB	87.2%
	Layer 28	252.5 KB	89.6%
	Layer 41	590.1 KB	85.9%
	Layer 41	680.5 KB	87.1%
	Layer 41	688.7 KB	87.6%

First, we can notice that in several cases, the proposed model implies an improvement of the accuracy despite the constraints on memory size and intermediate data. This is more observable in the 3G case, when the communication constraint

Table 3.3: The accuracy of VGG16 and MobileNetV1 for different memory sizes at the edge with LoRa (25 KB) communication constraint. The white rows indicate that the model size can meet the MCUs capacity. We also indicate the accuracy and memory of the unmodified approach (Teacher T). This is the original accuracy without compression of the raw data.

	Splitting layer	Memory size at the edge	Accuracy
	No splitting (T)	553.16 MB	70%
VGG16	Layer 7	377 KB	63.9%
	Layer 11	1.0 MB	62.2%
	Layer 11	1.0 MB	67.2%
	Layer 11	1.1 MB	72%
	Layer 15	3.8 MB	67.2%
	Layer 15	4.3 MB	73%
	No splitting (T)	17.3 MB	86.5%
MobileNetV1	Layer 28	198.4 KB	76.2%
	Layer 28	282.5 KB	84.5%
	Layer 41	477.4 KB	85.5%
	Layer 41	571.6 KB	85.5%
	Layer 41	568.6 KB	86.7%

is less important. This improvement is due to knowledge distillation, which can reduce the size of the neural network while improving the accuracy thanks to transfer learning. Moreover, as previously observed in Fig.3.7, the joint end-to-end training of knowledge distillation and auto-encoder proposed in this paper allows better results.

Furthermore, we can observe that generally, the smaller the student capacity (which in turn produces smaller memory size), the lower the accuracy, but this is not always the case. We find the same unexpected results as previously: under 3G constraint, splitting MobilenetV1 at layer 11 gives a better accuracy than splitting it layer 15, even if the memory size of the network at the edge is smaller (78.1% accuracy with memory 1.1 MB when splitting at layer 11 / 77.4% with memory 4.2 MB when splitting at layer 15). This implies that even if we have an edge device with enough memory to run the 15 first layers, it's more optimal to run only the 11 first layers. As explained in previous sections, this is probably due to the fact that knowledge distillation is optimal when splitting the network in the middle layers.

However, we can observe that the optimal splitting layer is not same under different network constraints : with a higher bandwidth constraint (LoRa), a better accuracy is obtained when splitting the network at layer 15. Indeed, in a very constrained network, the effect of intermediate data compression is much higher. Therefore, cutting the network at higher layers, generating a smaller output, gives a better accuracy at the cost of higher memory requirements at the edge.

Furthermore, we can notice that the student architecture must be chosen carefully : in Table 3.3, two different student networks with the same memory size (1.1 MB) give a 4% difference in accuracy.

Finally, we can notice that, thanks to knowledge distillation, splitting the neural network at low layers enables to meet memory requirements of very constrained devices such as MCUs (as depicted in Table 2.1) with flash memory size ranging from 256 KB to 2 MB, while maintaining a good accuracy, or even improve it. Hence, thanks to the model proposed in this paper, it is possible to choose where to split the network in an optimal way given memory and/or bandwidth constraints.

The results show that the optimal splitting may vary according to the different constraints. For example, when running MobilenetV1 in a scenario where the network condition changes from high available bandwidth to saturated network, it is necessary to switch the splitting from layer 11 to layer 15 in order to keep an optimal accuracy, if the edge device possesses enough memory. Another example can be given in the case of a smartphone running several computationally intensive applications at the same time: depending on the applications which are running, more or less memory will be available so the choice of the splitting layer may differ over time.

3.6 Comparison with Related Works

In order to compare our approach with the related works, we performed comparison in these two categories:

3.6.1 Encode (E)

This category encodes the images directly before sending it to the cloud which is the classical approach (first approach in Fig.3.1). For this comparison, we compress the image by using JPEG compression (**E-JPEG**) as described in [Grulich and Nawab, 2018] and auto-encoder (**E-AE**) as seen in [Dong et al., 2020]. The

raw data is compressed to achieve a size of 25KB (LoRa) or 200KB (3G). The results are presented in Table 3.4.

Table 3.4: Comparison of CDE with E-JPEG and E-AE.

Model	Network	CDE	E-JPEG [Grulich and Nawab, 2018]	E-AE [Dong et al., 2020]
MobileNetV1	LoRa	**86.7%**	84.5%	45.9%
VGG16	LoRa	73%	**76.5%**	27%
MobileNetV1	3G	**90.3%**	86.3%	75.9%
VGG16	3G	**78.1%**	77.1%	46.4%

We observe that when we directly apply the auto-encoder at layer 0 to compress the image instead of compressing intermediate data, we obtained a lower accuracy. This could be explained by the fact that if we split the network, the output features are already pre-trained and suitable for the task. When we just compress directly the image with an auto-encoder, the auto-encoder is not exposed with the final objective (e.g. classification), so it is trained blindly without knowing what the actual objective is.

On the other hand, when we compress the features from the split network using an auto-encoder, the features are already suitable for the task. This is also the case when the image is compressed with JPEG, even if the difference of accuracy is less visible. This reinforces the importance of the proposed approach: splitting the network, then compressing with an auto-encoder and training with distillation.

3.6.2 Cut-Encode (CE)

In this category, the execution of DNN is split between the edge and the cloud while the intermediate output is encoded with the auto-encoder (**CE-AE**) to achieve a size of 25KB (LoRa) or 200KB (3G). Previous work that follows **CE-AE** paradigm includes [Jong Hwan et al., 2018]. [Emmons et al., 2019] also en-

codes intermediate data with lossy compression (quantization, huffman coding), and does not distil the edge part. The results are presented in Table 3.5.

Table 3.5: The trade-off between accuracy and memory size at the edge between our CDE and CE-AE [Jong Hwan et al., 2018].

Model	Network	CDE (Acc.)	CDE (Memory)	CE-AE (Acc.)	CE-AE (Memory)
MobileNetV1	LoRa	**85.5%**	**477.4KB**	83.3%	870.3KB
VGG16	LoRa	**73%**	**4.3MB**	67.6%	31.2MB
MobileNetV1	3G	85%	**438.8KB**	**85.1%**	832.4KB
VGG16	3G	**77.4%**	**4.2MB**	69.8%	31.1MB

We observe that, while for MobileNetV1 the accuracies of CDE and CE-AE are relatively the same, for VGG16 the knowledge distillation of the edge part allows an improvement of 6 to 8%. Moreover, this enables us to significantly reduce the memory size of the model at the edge, which is divided by 2 for MobileNetV1 and by 7 for VGG16.

3.7 Summary

This is the first approach proposing to split neural network inference between the edge and the cloud while optimising both bandwidth and memory constraints. For this, we proposed a novel method which jointly trains knowledge distillation at the edge and intermediate data compression with an auto-encoder. This approach allows to find the optimal partition under specific constraints. It makes it possible to execute part of the network on very constrained devices such as MCUs, and under poor network conditions (LoRa) while limiting the accuracy drop.

Moreover, we showed that the optimal splitting layers are not necessarily the upper ones, as suggested by previous work, but must be chosen carefully according to the constraints. This is a promising technique for applications with limited

bandwidth and memory size at the edge devices. It enables to have guidelines to choose the splitting layer depending on these constraints. This optimal splitting may vary over time according to the evolution of the constraints. Further memory reduction could be achieved with quantization, which is orthogonal to the proposed technique.

In terms of training, the proposed method allows to use a pre-trained CNN model as a basis. However, the results show that it is necessary to retrain the whole network after the auto-encoder insertion. It would be interesting to explore the opportunity to perform classification on compressed intermediate representation directly, rather than expanding it back to original size. It could allow to avoid the last fine-tuning step where the whole network is re-trained. However, removing this step could have a negative impact on accuracy, because our experiments showed that this fine-tuning is necessary to adapt the weights in the cloud part with regards to the student network which is in the edge.

In the next section, we design a framework to split the neural network inference between the cloud several edge nodes, while taking into account the heterogeneity of the input images, as a result of dynamic changes in the scene for example.

4 | Cloud/Multi-edge image dependent DNN patitioning

In this chapter, we extend the method proposed in Chapter 3 to the multi-edge case.

This chapter is organised as follows : Section 4.1 introduces the approach. In Section 4.2, we give an overview of the system design. Section 4.3 details the different stages of the proposed solution. Section 4.4 presents the evaluation of this approach, before concluding.

4.1 Introduction

As exposed in Section 2.4, different images have different classification difficulties, and using early-exits can greatly improve the DNN efficiency by allocating less computation effort to easy samples.

Moreover, in practice, it is becoming more common to find multi-camera networks for functional needs (e.g. monitoring several places or capturing views from different angles) [Aghajan and Cavallaro, 2009, Ryan et al., 2014, Yang et al., 2022]. These are characterized by having short-range, high-speed links (e.g. WiFi) and long-range, low-speed links (e.g. cellular or LoRA). In recent years, an increasing development of distributed computing has been observed in the IoT context : with the advances of semiconductor technology, it is possible to embed computation on end-devices, especially cameras [Chien et al., 2015]. Indeed, as explained in Section 1.1, in environments where the network is severely limited, the cloud-only solution is not viable. This is the case in several application scenarios (camera networks deployed in tunnels [Torres-Zapata et al., 2019], smart agriculture [Shadrin

et al., 2020, Segalla et al., 2020, Monteiro et al., 2021] or animals observation in nature reserves located in remote areas [Zualkernan et al., 2022]).

In order to be less dependant on the cloud availability in these contexts, there is a need to find a solution allowing a distributed processing between the end-nodes. More precisely in the deep learning context, the challenge is to distribute the DNN inference between multiple nodes and the cloud.

A promising approach is to integrate early exits into the neural network [Matsubara et al., 2022]. This makes it possible to obtain a prediction directly at the local level even if the network is saturated, and to leave the most complex images to the cloud. However, most of the existing early-exit approaches are designed to be executed on a single platform (cloud) [Teerapittayanon et al., 2017b, Lo et al., 2017, Neshatpour et al., 2019, Wang et al., 2019, Li et al., 2019, Phuong and Lampert, 2019, Elbayad et al., 2019] (Figure 4.1 (a)), or on a dual platform with one single edge computational element and the cloud [Zeng et al., 2019, Laskaridis et al., 2020a, Wolczyk et al., 2021] (Figure 4.1 (b)), whereas including several early-exits on a single computing device is limited by the memory size of the edge devices.

To the best of our knowledge, few existing works have explored the distribution of a neural network between the cloud and several computing elements at the edge (Section 2.5). Thus,in this chapter, we propose the first approach allowing the distribution of a neural network between the cloud and multiple devices at the edge, with several splitting points. This is done by inserting multiple early-exits at the edge level (Figure 4.1 (c)). The proposed approach allows to optimise both accuracy and execution time. The computing elements can be smart cameras or any other device that embeds computing capabilities. They are referred as "processing elements (PE)" or "clients". The clients can communicate through a local

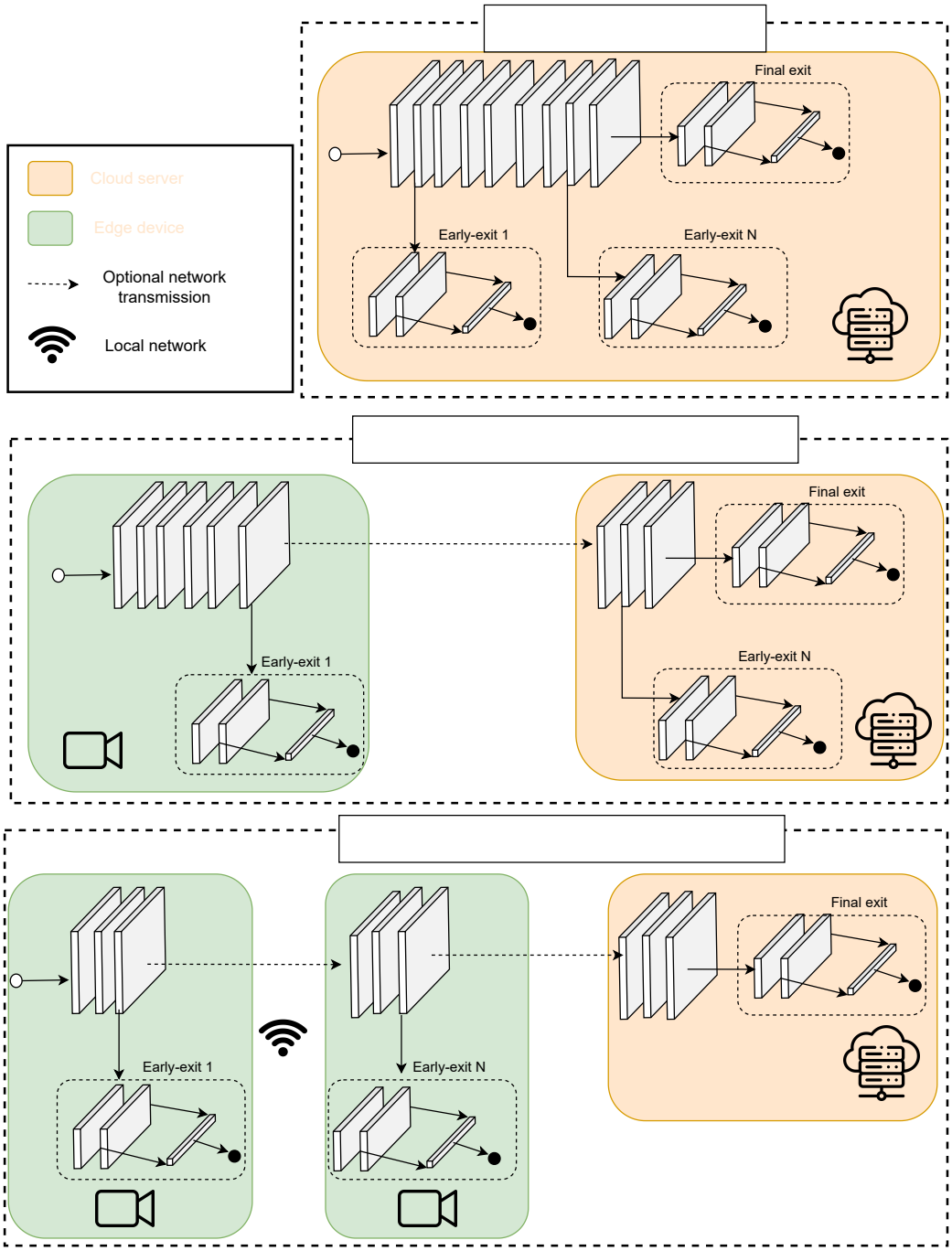


Figure 4.1: Comparison between different early-exit methods

network (Wifi), whereas the cloud can be accessed via limited bandwidth (LoRa).

This involves several challenges. Indeed, for each Processing element (PE), the question arises of whether to stop the inference or to continue the calculations by sending the data to the next processing element. Moreover, since the intermediate data of neural networks are often large, it is crucial to find effective ways to compress them to avoid high latency.

In order to address these issues, our contributions are as follows:

1. We propose the first approach which distributes neural network execution between a cloud server and multiple clients at the edge while optimising both accuracy and time.

2. A training method to adapt the topology of the network to its progressive execution. This consists of inserting early-exits and compression bottlenecks (auto-encoders) to reduce the amount of communication whilst limiting the loss of precision.

3. A multi-objective framework to distribute the workload between the different processing elements. This framework includes an offline and an online stage.

Note that we consider a single DNN distributed over multiple edge nodes and the cloud, i.e. only one node is capturing images. Indeed, modern surveillance cameras [Arlo, 2023, Zumimmal, 2023] and camera trap systems [Klemens et al., 2021] are often equipped with motion-detection or thermal triggering features. The computing capabilities of these devices are not necessarily used. The distribution of the DNN models on several devices overcomes this problem to take full advantage of all the available computing resources, and this is the first step towards a possible multi-sensor approach.

4.2 System overview

4.2.1 General description

The proposed approach is divided into three main steps, as illustrated in Figure 2 :

- **Step 1 : Architecture deployment and training (Offline):** The first step consists in optimising and adapting the architecture of the DNN to its progressive execution. It splits the DNN into several parts according to the number of end devices. This process includes the insertion of early-exits and bottlenecks in the neural network architecture. It is performed offline, at compile time.
- **Step 2 : Workload sharing algorithm (Offline):** During this step, we share the workload between the different early-exits according to user-defined objectives in terms of time and accuracy. This is done using linear programming optimisation. The output of the linear program (LP) (linear program) is the proportion of images that must be output at each exit. This step is performed offline.
- **Step 3 : Distributed execution (Online):** The last step is the DNN execution by the PE and the cloud on real data. The workload distribution is adapted at runtime to match the distribution given in step 2.

4.2.2 Metrics

With N the number of early-exits and $i = 1, 2, ..., N+1$ an integer variable ($N+1$ represents the final exit, which is in the cloud), we defined the following metrics:

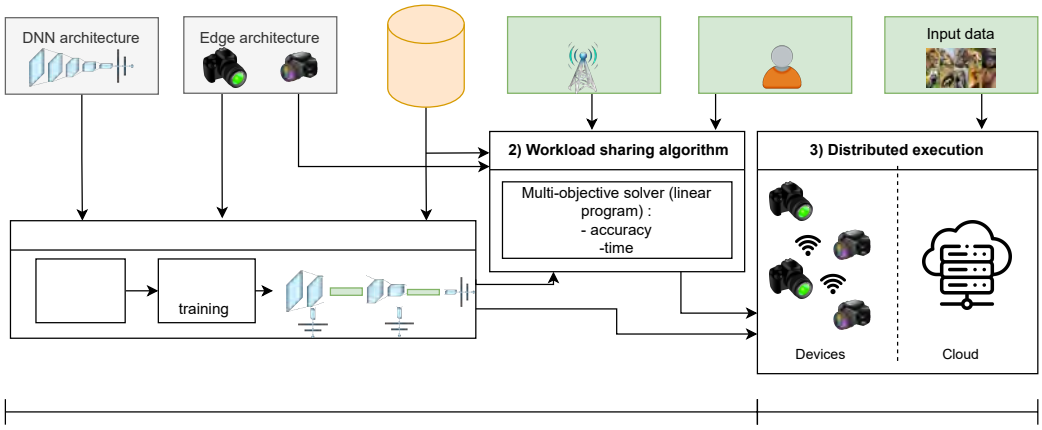


Figure 4.2: System overview

4.2.2.1 Accuracy

- *Accuracy of early-exit i over all images ($AccAll_i$)* : this is the accuracy of early-exit i computed over all images, i.e. when all images are output at early-exit i. We use this accuracy during the training process (step 1 and 2). In the same way, $ErrAll_i$ is the error percentage of early-exit i computed over all images.

- *Accuracy of early-exit i (Acc_i)* : this is the accuracy of early-exit i computed only over the images output at this exit, i.e. when the workload is shared between the different early-exits, at runtime (step 3). In the same way, Err_i is the error percentage of early-exit i computed only over the images output at this exit at runtime.

- *General accuracy (Acc_G)* is the overall accuracy of the neural network computed over all images, with workload distribution between the early-exits (step 3). In the same way, the overall error percentage (Err_G) is the overall error percentage of the neural network.

4.2.2.2 Time

- *Computation time of early-exit i ($CompT_i$)* : this the computation time required to execute the neural network from early-exit $i-1$ to early-exit i, including early-exit i's layers. For the first early-exit, this is the time required to execute the neural network from input to early-exit 1, including early-exit 1's layers.
- *Communication time between early-exit $i-1$ and early-exit i ($CommT_{i-1,i}$)* : This is the time needed to transfer intermediate representation from early-exit $i-1$ to early-exit i. It is determined by the size of the intermediate representation and the network throughput.
- *Total execution time of early-exit i ($ExcT_i$)* : this is the total time required to process an image until early-exit i. It consists of the computation time of all previous exits and the communication times between them. We have :

$$ExcT_i = \sum_{j=1}^{i} CompT_j + CommT_{j-1,j} \tag{4.1}$$

4.3 Detailed design

4.3.1 Step 1 : DNN architecture deployment and training (Offline)

To adapt the neural network to its progressive execution, it is necessary to transform it. The different transformations that are performed on the initial neural network architecture are the following :

- **Neural network splitting** : The system first splits the neural network into a number of parts which is equal to the number of edge devices plus one (cloud). The choice of the splitting points is guided by three principles (as seen in Chapter 4) : 1) the fully-connected layers' heavy computations are left to the cloud, 2) the different parts must fit into the edge devices' memory, 3) the splitting points are chosen where the size of the intermediate data is smaller (after the Max-Pooling layers). This choice is done empirically and the design of an algorithm enabling to explore all possible options and choose the optimal splitting points is left for future work.

- **Bottlenecks insertion** : We insert bottlenecks in the DNN to reduce the size of the intermediate data at each splitting point. Each bottleneck is an auto-encoder composed of a convolution layer which reduces the size of the data by a shrinking factor f (contraction layer), followed by another convolution which restores the size of the data (expansion layer). Adding only two convolution layers limits the computation overhead. Limiting the accuracy loss and the computation overhead is important since several bottlenecks are inserted into the neural network. We use a factor of $f = 1/2$. It allows to divide the size of the intermediate representation by 2 while limiting the total accuracy loss to 2.5%. We also tested $f = 1/4$ and $f = 1/8$ but the accuracy loss was too important (7.3% and 11.5% respectively). We train all the bottlenecks together using cross entropy. Then, we use NPZ compression [NumPy, 2023] to further reduce the data size.

- **Early-exits insertion** : The early-exits are positioned just before the bottlenecks. Their architecture is adapted to their depth in the neural network [Huang et al., 2017b]. Then, all early-exits are jointly trained along with the bottlenecks, which are fine tuned. We define a loss function for each of the classifiers and minimize the weighted sum of cross-entropy

losses per sample as follows:

$$\mathcal{L}_{joint}([\hat{y}^1, \ldots, \hat{y}^{N+1}], c) = \sum_{j=1}^{N+1} \lambda_j L_{CE}(\hat{y}^j, c) \tag{4.2}$$

where $[\hat{y}^1, ..., \hat{y}^{N+1}]$ indicates outputs from $N+1$ classifiers, and the correct label c is shared across all the classifiers in a model.

4.3.2 Step 2 : Workload sharing algorithm (offline)

Once the neural network architecture is fixed, we have to determine the early-exit strategy, i.e. to decide which images will be output at which early-exit. We present below two possible strategies: Fixed threshold (FT) and linear programming optimisation.

4.3.2.1 Fixed-threshold strategy (FT)

A CNN generally consists of convolutional layers, pooling layers and fully-connected layers. A softmax function layer is usually applied to the final layer of a CNN. In a multi-exit network, each classifier has a softmax layer, which is defined as:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4.3}$$

where z_i is the output of the final fully-connected layer for the $i-th$ label, K the total number of labels, and j the classifier index. This softmax function outputs a K-dimensional vector y of real values between 0 and 1. Softmax outputs are generally interpreted as indicators of the extent of CNNs confidence for different output classes. As a result, they can serve as a measure of confidence levels [Guo et al., 2017, Laskaridis et al., 2020a].



Figure 4.3: Same class images output by different exits.

For this, we define a confidence threshold th_{conf} and we estimate a classifier's confidence for a given input using the top-1 output value of its softmax layer. An input takes the $i-th$ exit if the prediction confidence is higher than th_{conf}, i.e. if we have:

$$arg_i\{\underset{i}{max}\{softmax_i\} > t\} \tag{4.4}$$

Figure 4.3 is a visualisation some examples of softmax outputs for different classes of CIFAR-100 dataset when inserting 4 early-exits in a VGG16 DNN ($th_{conf} = 0.7$). We can see that the softmax vales allows to apprehend the level of difficulty of the images, even inside a same class. For example, images with several objects have a higher level of difficulty than images with a single object. Therefore, they need more computation and are output at later early-exits. This phenomenon is visible for lemon and pepper classes. Another example is the zoom a level : an important zoom level implies a higher difficulty (alligator class).

The confidence threshold th_{conf} is tunable and provides a trade-off between the accuracy and the latency of the progressive model. If the system fixes a high confidence threshold, a classifier will only classify an image if it is confident enough in its prediction. Otherwise, it will pass it to the next classifier, implying network communication.

This trade-off can also be apprehended in terms of True positive (TP), False positive (FP) and F1-score, as illustrated in Tables 4.1 and 4.2.

Table 4.1: True Positives (TP) and False Positives (FP) scores for VGG16 with four exits and different thresholds

Threshold	First exit		Second exit		Third exit		Fourth exit	
	TP	FP	TP	FP	TP	FP	TP	FP
10%	59%	40%	65%	34%	69%	30%	69%	30%
20%	58%	35%	64%	31%	69%	29%	69%	30%
30%	55%	26%	62%	25%	68%	26%	68%	29%
40%	50%	18%	59%	19%	67%	22%	68%	27%
50%	45%	11%	55%	13%	64%	17%	67%	24%
60%	40%	7%	51%	9%	61%	13%	65%	21%
70%	34%	4%	46%	5%	57%	9%	63%	17%
80%	27%	2%	40%	3%	52%	6%	61%	14%
90%	19%	07%	32%	1%	45%	3%	58%	10%

Table 4.2: F1 scores and accuracies for VGG16 with four exits and different thresholds

Threshold	First exit		Second exit		Third exit		Fourth exit		Final exit		
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc
10%	56.05%	0.56%	0.05%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	56.10%
20%	55.20%	0.57%	1.90%	0.03%	0.26%	0.00%	0.02%	0.00%	0.01%	0.00%	57.39%
30%	52.23%	0.57%	5.95%	0.10%	1.66%	0.03%	0.20%	0.00%	0.03%	0.00%	60.07%
40%	47.95%	0.55%	9.90%	0.16%	4.13%	0.07%	1.14%	0.02%	0.21%	0.00%	63.33%
50%	42.68%	0.51%	13.67%	0.22%	7.01%	0.11%	2.57%	0.04%	0.54%	0.01%	66.47%
60%	37.25%	0.47%	15.88%	0.25%	9.50%	0.16%	4.33%	0.07%	1.46%	0.02%	68.42%
70%	31.84%	0.42%	17.13%	0.27%	11.23%	0.18%	6.71%	0.11%	2.92%	0.05%	69.83%
80%	25.29%	0.35%	18.07%	0.28%	12.95%	0.21%	9.39%	0.15%	4.95%	0.08%	70.65%
90%	17.53%	0.25%	17.62%	0.27%	15.28%	0.25%	12.37%	0.19%	8.66%	0.13%	71.46%

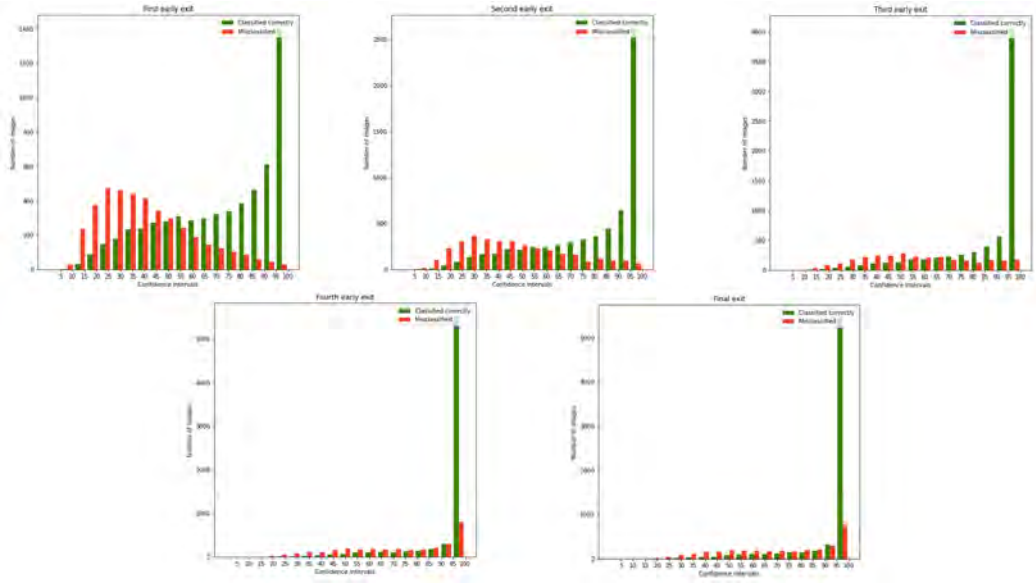


Figure 4.4: Distribution of confidence scores at each exit

Figure 4.4 shows the the distribution of the confidence scores at each exit. It shows the limits of the confidence threshold strategy, with two types of problematic cases :

- Cases of true positives which have a low confidence score
- Cases of false positives which have a high confidence score

Therefore high confidence threshold will imply a higher accuracy but also a higher latency. This phenomenon is even more pronounced in a highly distributed scenario where all classifiers are distributed between different devices. Hence, it is important to carefully chose the workload sharing strategy.

4.3.2.2 Linear programming optimisation (LP)

We propose an alternative to the fixed-threshold strategy to distribute the workload. The goal is to introduce more flexibility in the accuracy/latency trade off. We formalise this problem as a linear program with the objective to jointly opti-

mise accuracy and latency. The output of the linear program is the proportion of images that must be classified by each early-exit.

The choice of the linear program model is due to the optimal properties of such models and the availability of powerful resolution libraries such as CPLEX [IBM-Corp., 2023] . In general, this type of mathematical formulation is not recommended for problems with large instances because even if the software often makes it possible to solve large instances, the resolution time is not guaranteed for problems with thousands of variables and constraints. However in our case, the number of early exits is very limited (generally less than a dozen), making the number of controllable variables quite small. This leads to fast and accurate solutions of the proposed LP model.

The linear program model is detailed below.

Controlled parameters With N the number of early-exits, the controlled parameters of the linear programs are:

- $NBIm_i, i = 1, \ldots, N+1$: number of images classified by early-exit i
- $S_i, i = 1, \ldots, N$: $S_i = 1$ if the early-exit i is used and $S_i = 0$ otherwise. This parameter allows to activate/deactivate classifier i.

Non-controlled parameters The non-controlled parameters of the linear program are :

- $ErrAll_i$: Error percentage of output i as defined in Section 4.2.2
- $ExcT_i$: Execution time for an image taking output i as defined by equation 4.1
- $NbImT$: total number of images

- BI : big integer

LP model The objective the linear program is to minimise error percentage and latency :

$$\begin{aligned} \text{minimize } W_{time} * \sum_{i=1}^{N+1}(NBIm_i * ExcT_i) \\ +W_{err} * \sum_{i=1}^{N+1}(NBIm_i * ErrAll_i) \end{aligned} \tag{4.5}$$

$$\begin{aligned} 0 \leq W_{err} \leq 1 \text{ (precision weight)} \\ 0 \leq W_{time} \leq 1 \text{ (execution time weight)} \end{aligned} \tag{4.6}$$

subject to

$$\sum_{i=1}^{N+1} NBIm_i = NbImT \tag{4.7}$$

$$NBIm_i \leq BI * Si. \forall i \tag{4.8}$$

$$BI * NBIm_i \geq Si. \forall i \tag{4.9}$$

$$W_{err} + W_{time} = 1 \tag{4.10}$$

The weights W_{time} and W_{err} allows to give more importance to accuracy or latency according to the application requirements.

Constraints 4.8 and 4.9 ensure consistency between the opening/closing of an EE and the number of inferences that will be assigned to it. The constraint 4.8 ensures that if no inferences are assigned to an early exit, then that EE should not exist. Here, BI is a large enough constant that ensures that the left-hand side of the inequality is less than or equal to 0 when no inferences are assigned to EE_j. Therefore, if no inferences are assigned to EE_j, the binary variable Si must be

equal to 0, which means that the EE must be closed. This constraint ensures that warehouses are not kept open unnecessarily, which would result in additional costs.

Note that in the objective function, depending on the DNNs and the execution platforms, the weights of the two optmisation criteria (execution time and precision) must be adjusted so that the two parts of the addition are of the same order of magnitude. Indeed, if the values on both sides of the addition are too unbalanced, then the model could favor the criterion giving the greatest value even if its weight, in the objective function, is lower.

As precision and execution time are very different metrics, the choice of weights assigned to each of these criteria in the objective function must be made with caution. If such a choice is easily made by an expert in the application domain, it can nevertheless prove relatively tedious for others. Thus, the way that seems to be the safest and most efficient to set these parameters is to rely on several executions, a priori, of the ILP (with various values for the weights) and to observe the Pareto front of the set of solutions. This makes it possible to observe variations in the solution as a function of the weights and therefore choose the appropriate values.

4.3.3 Step 3: Distributed execution - Dynamic Threshold Configuration (Online)

The last step of the optimization approach consists in dynamically modifying the confidence thresholds at runtime in order to match the output percentages given by the linear program. This is *Dynamic Threshold Configuration (DTC)*. This is done with the aim of adapting the confidence thresholds as well as possible to the properties of the real images processed during the DNN execution. Indeed,

these images can sometimes be significantly different from those used for network training and for the previous optimization steps.

The principle used for the dynamic modification of the thresholds is relatively simple. The idea is that when we reach the image output percentage on an early exit, we consider that the optimization criteria set in the linear program have reached the optimum. Therefore, we increase the threshold corresponding to this output by a given percentage Δ_{th} to increase the precision without altering the objective. This procedure is repeated every Δ_{images} processed images. The threshold increasing Δ_{th} and the number of Δ_{images} images are chosen empirically and can quite easily be configured and adapted to each application context.

4.4 Evaluation

4.4.1 Implementation details

We implemented our model using Keras framework and the model was trained on GPU from Google Colab. For all experiments, we used 100-sports classification dataset [Gerry, 2023] with 14572 images, 100 classes, and a medium image definition (224×224). To demonstrate the effectiveness of the proposed system, we employed VGG16 [Simonyan and Zisserman, 2014] with the baseline accuracy 85.3% in the used dataset. In the first training stage, we trained the bottlenecks with Adam optimizer [Kingma and Ba, 2014] for up to 64 epochs. We used 1e-3 as learning rate. In the second stage of training, we jointly-trained the early-exits and fine tuned the bottlenecks as described in section 4.3.1. We used Adam optimizer with learning rate 1e-4 and up to 30 epochs. For both training stages, we divided 20% of the training data as the validation set and used 32 as the batch size.

The application consists in a DNN with three early exits and one final exit (cloud side). It was tested on NVIDIA V100 Tensor Core GPU for the cloud part and on Intel(R) Xeon(R) CPU 1.60GHz for the edge devices. Communications, in our experiments, use two protocols : a 30 kbps LoRa between the third early exit and the cloud, and a 1.3 Gbps Wifi between the edge devices.

4.4.2 Architecture deployment and training (Step 1)

We used two different baselines to evaluate the efficiency of our architecture and training methods (step 1) :

- *No bottlenecks (**NB**) [Teerapittayanon et al., 2017a] :* In this solution, the size of the intermediate data sent between the different early-exits is not reduced.
- *Separate training (**SL**) [Xin et al., 2020] :* In this solution, the early-exits are trained one-by-one separately, while the other weights in the rest of the network are frozen, including the bottlenecks, which are not fine tuned.
- *Cloud-only (**CO**) [Grulich and Nawab, 2018] :* This is the classic and 'naive' solution, consisting in allowing one single exit in the DNN at the cloud level. All images are offloaded to the cloud.

We compare these two methods with the joint training method (**JL**) that is used in the proposed system. The size of intermediate data is presented in Table 4.3, and the accuracy of the early-exits is presented in Table 4.4.

We notice that the data compression with bottlenecks and NPZ compression allowed a reduction of 83%, 85% and 79% respectively of the size of the data leaving early-exit 1, 2 and 3, as showed in Table 4.3. Moreover, this compression involved an accuracy loss smaller than 3%, as shown in Table 4.4.

Table 4.3: Memory size of the early-exits' output when using different architectures and training methods.

Exit #	NB	SL	JL
EE1	1.02 MB	171 KB	171 KB
EE2	349 KB	50.8 KB	50.8 KB
EE3	81.8 KB	16.5 KB	16.5 KB

Table 4.4: Accuracy of the early-exits when using different architectures and training methods (all images are output at each early-exit).

Exit #	NB	SL	JL	CO
EE1	57.4%	61.6%	72.2%	/
EE2	80.8%	68.2%	73.4%	/
EE3	82.6%	75.2%	75.4%	/
FE (Cloud)	85.3%	82.8%	82.8%	85.3%

Furthermore, Table 4.4 shows that the Joint learning (JL) training method is beneficial in terms of accuracy, especially for early-exits which are shallower, with a gain of 9.6% for EE1 and 5.2% for the EE2 compared to Separate learning (SL). This allows to reach high accuracy in the earliest early-exits with 72.2% and 73.4% for EE1 and EE2 respectively, and to favor local computation.

Interestingly, implementing the bottleneck on the first node (EE1) actually significantly improves the accuracy of the exit classifier, probably by reducing its overfitting by reducing the information density in the feature vector.

4.4.3 Workload sharing algorithm (Step 2)

In this section, we compare the two workload sharing policies presented in section 4.3.2 : fixed confidence threshold and linear programming optimisation. The linear programming formulation of the problem aims at optimizing two criteria: overall execution time and accuracy. Weights are associated to these two criteria

in the objective function to favor one or the other of the criteria according to the application needs.

Figure 4.5 shows the percentage of images taking each early exit according to the LP results, and according to a fixed threshold policy with confidence thresholds of 0.3, 0.5, 0.7 and 0.9. We notice that, as expected, higher thresholds involve more images offloaded to the cloud.

In the LP model we considered equal weights for error rate (accuracy) and execution time ($W_{time} = W_{err} = 0.5$). This means that neither of the two criteria is favored over the other. The results show that the three early exits are allowed in the DNN, in addition to the Cloud exit. According to the LP, more than half of images take the first early exit and only 7% take the last early exit (before the cloud). Such results are quite predictable in the sense that the communication between the EE3 and the cloud is very expensive in terms of time.

We can see that the threshold policy does not allow flexibility in the use of the early-exits. For example, the 0.9 confidence threshold implies that 33% of images are offloaded to the cloud while only 9% images are inferred by EE3, which seems underused. The linear programming policy allows a more balanced workload sharing, as it aims at optimising both accuracy and latency. It favors local computation and minimises cloud offloading.

4.4.4 Distributed execution (Step 3)

4.4.4.1 Dynamic threshold configuration

We execute the DNN with a new images dataset representing the real life images. It is a subset of the 100-sports classification dataset containing 7000 images that were not used for the training. We initially set thresholds corresponding to all

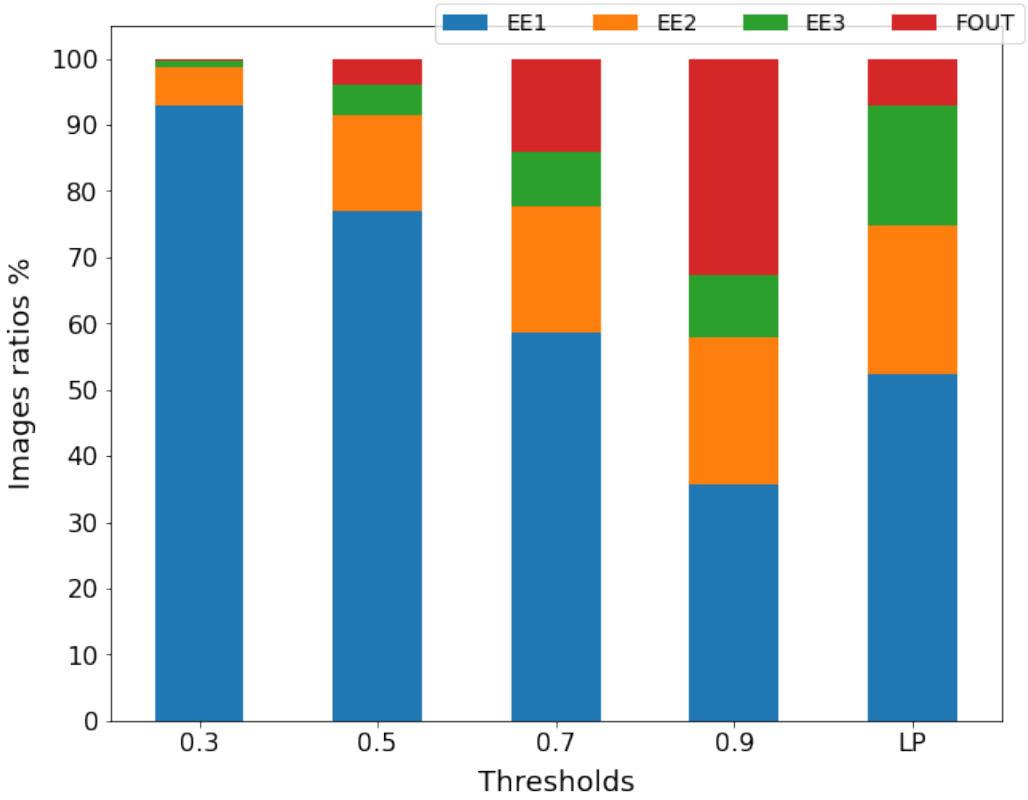


Figure 4.5: Percentage of images exiting at each exit with different workload sharing methods: Fixed threshold 0.3, 0.5, 0.7, 0.9 and Linear Programming. FOUT refers to the final exit (in the cloud).

early exits to 0.70. We integrated a function that checks the percentage of images taking each early exit every 500 processed images. The function increases a given threshold by 10% if the exiting images percentage is greater than the value given by the LP resolution, and decrease it by 10% otherwise. We chose this value because during our experimentation, it turned out that this is the value where the variation of the threshold begins to have a noticeable impact on the number of images exiting through an early exit.

Tables 4.5 and 4.6 show the percentage of images taking each early exit, the overall error rate and the overall average execution time per image (including computations and communication) according to the LP results before and after DTC.

Table 4.5: Percentage of images taking each early exit with linear programming workload sharing, before and after dynamic threshold configuration

Exit #	Before DTC	After DTC
EE1	52.4%	57.2%
EE2	22.4%	23.8%
EE3	18%	13.8%
FE (Cloud)	7.2%	5.2%

Table 4.6: Overall error rate and the overall average execution time per image (including computations and communication) according to the LP results before and after DTC

	Before DTC	After DTC
Avg. exec. time	0.245 s	0.22 s
Err. rate	19.6%	19.1%

Table 4.6 shows that after dynamic threshold configuration, the obtained average execution time per image is 0.2199 seconds (10 % lower than what provided by the LP) and the error rate remained almost unchanged (19.1). Furthermore, the solution obtained with the dynamic threshold configuration during run time gives results which are relatively close to those of the linear program in terms of frame rate taking each early exit. However, we can note an increase (respectively a decrease) in this rate on the first two exits (respectively the last two). This is explained by the fact that the real images include a concentration of "easy" images at the instants immediately following the thresholds update. Indeed, such a configuration forces the DNN to output a lot of images at the first early exit and quickly reach the rate given by the linear program. During the rest of the execution time, until the next thresholds update, some 'very easy' images still take EE1, even if

it has already reached its rate. This same phenomenon occurs for EE2. Therefore the last two exits cannot reach the frame rate given by the LP.

Such a behavior necessarily gives a better overall execution time because the first two early exits are the least greedy in terms of execution time and communication. This solution could have lowered the accuracy of the system (increased the error rate), but in our experiment the accuracy was relatively stable. This can be explained by the fact that the real images were globally a little less complex than those of the test dataset.

4.4.4.2 When and how the LP and Dynamic Threshold should be used?

The LP model is used to determine the optimal percentages of images exiting the CNN at each early exit in order to initialize the dynamic threshold variation algorithm. Thus, the LP is executed offline only once and this makes it even more interesting given the low overhead it generates in terms of execution time and computing resources use. The dynamic threshold variation algorithm is executed during runtime and its role is to force convergence towards the percentages obtained by the LP at each iteration.

4.4.4.3 Time/Accuracy analysis

Figure 4.6 is a time/accuracy graph comparing different workload sharing policies and training methods :

- Cloud-only (CO)
- Linear program with joint-learning (LP+JL)
- Fixed-threshold with joint-learning (FT+JL)
- Fixed-threshold with separate learning (FT+SL)

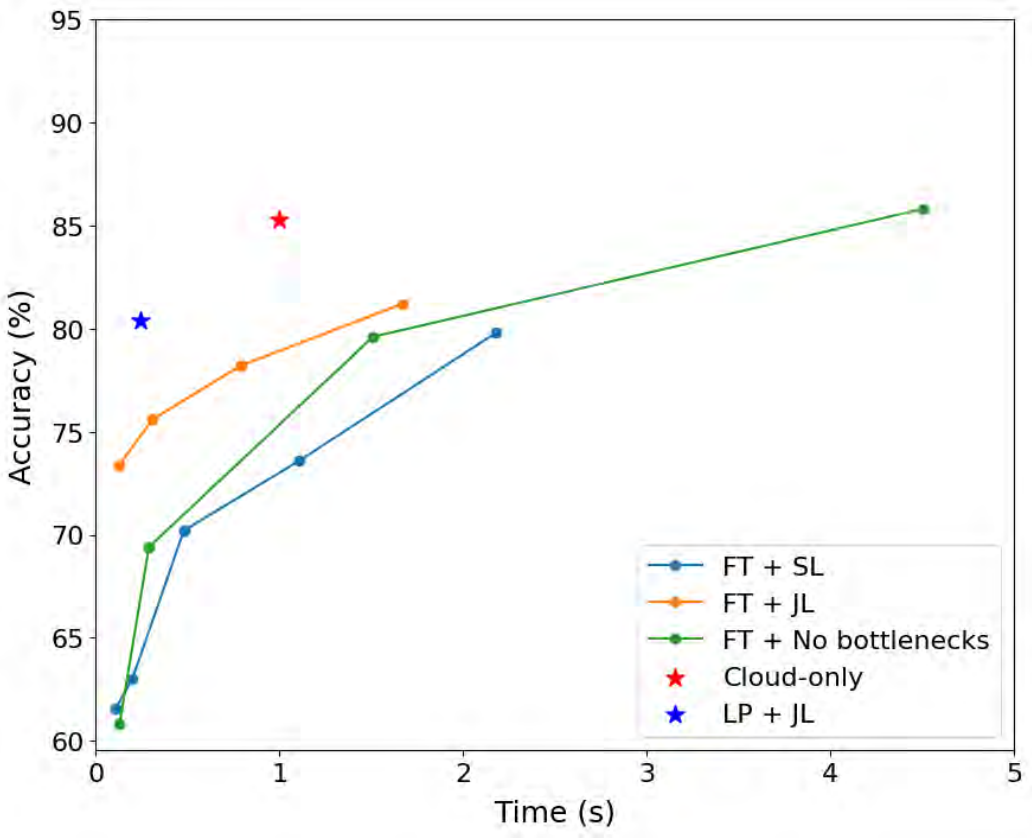


Figure 4.6: Time/Accuracy graph for different methods : 1) Fixed Threshold and Separate Learning (FT+SL), 2) Fixed Threshold and Joing Learning (FT+JL) 3) Fixed Thresholds and No bottlenecks (FT+NB), 4) Linear Programming + Joint Learning (LP+JL) 5) Cloud-only. For the fixed threshold methods, the confidence thresholds that are represented are 0.3, 0.5, 0.7 and 0.9. The times are the average time per image

- Fixed-threshold with no bottlenecks (FT + No bottlenecks)

First, the results show that the early-exit distributed execution approach, with linear programming workload sharing policy, allows a 80% gain in average execution time compared to the traditional cloud-only solution, with an accuracy drop limited to 4.4%. The high accuracy can be explained by the fact that the accuracy of each EE computed only on the images exiting at this EE is higher than the accu-

racy computed over all images. EE1 and EE2 in particular achieve high accuracies on the 'easy' images which are assigned to them.

Furthermore, we observe that the workload sharing policy plays an important role in the accuracy/time trade off. The LP allows a 5.3% gain in accuracy for a similar execution time of 0.2 s compared to the fixed threshold policy with joint learning and a confidence threshold of 0.5. With a 0.9 threshold, it is possible to reach a higher accuracy, but this is done to the detriment of time (1.6 s).

Regarding architecture and training methods, we notice without surprise that the no-bottlenecks architecture implies higher accuracy, but with an explosion of execution time. Furthermore, we observe that joint learning reaches higher accuracy than separate learning, as already suggested by the results of section 4.4.2.

4.4.4.4 Convergence

As the thresholds are dynamically adapted, both downwards and upwards, during the run time, the final result in terms of number of images taking each early exit ends up, in almost all cases, by converging towards the rates given by the linear program (with certain thresholds equal to 1 or 0 over long periods if necessary). However, the results obtained, in terms of execution time, may be relatively far from the optimum in the following specific cases:

- Real images with distributions of complexity levels too far from those of the images belonging to the test dataset.
- Real images with distributions of complexity levels close to those of the images belonging to the dataset, but with densities of occurrences that are too different. This is for example the case where the dataset images are uniformly distributed (in their arrival order) according to their complexity, and that, for the real images, the 'simple' images all appear at the beginning

and the complex images all at the end. This could correspond, for example, to alternating day and night.

In the two above mentioned cases, the problem comes from the data used during the test phase. It is for this reason that the training dataset must imperatively be chosen meticulously in order to take full advantage of the potential of our optimization approach. The proposed approach provides the best possible results in the case where the real images are partitionable into sets (continuous in time) all having complexity distributions close to that of the test images.

4.5 Summary

In this chapter, we presented a distributed progressive inference approach that addresses the challenge of partitioning DNN inference across multiple devices and the cloud. By employing an efficient workload sharing optimisation approach and a specific DNN architecture, this system is able to deliver high accuracy under strict latency constraints. However, energy consumption is not considered. In the next chapter, we extend this platform to systems where energy consumption is more important than time.