
# DNN ARCHITECTURES FOR RESOURCE-CONSTRAINED DEVICES AND LATENCY-SENSITIVE APPLICATIONS

Mateus da Silva Gilbert

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Miguel Elias Mitre Campista
Marcello Luiz Rodrigues de Campos

Rio de Janeiro
Fevereiro de 2024

DNN ARCHITECTURES FOR RESOURCE-CONSTRAINED DEVICES AND LATENCY-SENSITIVE APPLICATIONS

Mateus da Silva Gilbert

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientadores: Miguel Elias Mitre Campista
Marcello Luiz Rodrigues de Campos

Aprovada por: Prof. Miguel Elias Mitre Campista
Prof. Marcello Luiz Rodrigues de Campos
Prof. Marcelo Dias de Amorim
Prof. Daniel Sadoc Menasché

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2024

Gilbert, Mateus da Silva

DNN architectures for resource-constrained devices and latency-sensitive applications/Mateus da Silva Gilbert. – Rio de Janeiro: UFRJ/COPPE, 2024.

XIV, 70 p.: il.; 29, 7cm.

Orientadores: Miguel Elias Mitre Campista
Marcello Luiz Rodrigues de Campos

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2024.

Referências Bibliográficas: p. 65 – 69.

1. DNNs. 2. resource-constrained devices. 3. latency-sensitive applications. I. Campista, Miguel Elias Mitre *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

# Acknoladgement

One does not receive a Master of Science degree working alone. Firstly, I would like to thank my family for their unconditional support over the years. Long before I started my graduate journey, it was my parents' and relatives' support that brought me to where I'm standing now. Next, I would like to thank the professors that I have met in my undergraduate and graduate studies, whose work inspired me to pursue a research career. Specifically, I would like to thank my co-advisors who worked alongside me on this two-year journey, helping me in my research and in honing my skills. Also, I would like to thank the professors who made themselves available to compose the evaluation panel of this work. Last but not least, I would like to thank my colleagues with whom I've shared my academic life, since my undergraduate period. Their support helped me overcome the many hurdles one faces when pursuing an academic degree, whether it was working on projects or studying for tests together.

I would also like to thank the institutions that funded my research, namely CNPq, FAPERJ, FAPESP, CAPES. These public institutions were crucial in enabling my full-time commitment to my research by directly funding me or providing funds to purchase equipment.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

# DNNS PARA DISPOSITIVOS LIMITADOS E APLICAÇÕES SENSÍVEIS À LATÊNCIA

Mateus da Silva Gilbert

Fevereiro/2024

Orientadores: Miguel Elias Mitre Campista
Marcello Luiz Rodrigues de Campos

Programa: Engenharia Elétrica

Redes Neurais Profundas (Deep Neural Networks – DNNs) se tornaram ferramentas relevantes e populares para uma variedade de problemas que envolvem o processamento de grandes volumes de dados e respostas inteligentes. Graças à sua abordagem voltada a dados, DNNs oferecem desenvolvedores uma estrutura de trabalho relativamente simples que envolve a construção de modelos mediante interações com dados. Além disso, DNNs são estado da arte em aplicações que envolvem visão computacional (Computer Vision – CV) e processamento natural de fala (Natural Language Processing – NLP). Desenvolvedores podem recorrer a esses modelos bem estabelecidos para classificar imagens, rastrear objetos, melhorar a qualidade de imagens, processar escrita, gerar textos etc., seja como aplicação principal ou auxiliar. Porém, implementar DNNs em dispositivos com limitações de recursos e cenários sensíveis a latências pode ser uma tarefa difícil, especialmente quando esses modelos são construídos com múltiplas camadas e parâmetros. Esta dissertação de mestrado discute essas dificuldades de implementação e apresenta dois modelos de NN cujos objetivos são aliviar esses problemas: Autoencoders Assimétricos (Asymmetric Autoencoders – AAEs) e DNNs com saídas antecipadas (Early-Exit DNNs – EE-DNNs) para segmentação semântica. O primeiro modelo oferece uma arquitetura mais adequada a dispositivos com recurso limitados. O segundo busca estender o sucesso das EE-DNNs com classificação de imagens para segmentação semântica. No geral, esta dissertação se propõe a discutir a importância de incorporar mudanças na arquitetura de DNNs para facilitar sua adoção em dispositivos com recursos limitados e aplicações sensíveis à latência.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DNN ARCHITECTURES FOR RESOURCE-CONSTRAINED DEVICES AND LATENCY-SENSITIVE APPLICATIONS

Mateus da Silva Gilbert

February/2024

Advisors: Miguel Elias Mitre Campista
Marcello Luiz Rodrigues de Campos

Department: Electrical Engineering

Deep Neural Networks (DNNs) have become a relevant and popular tool to handle a variety of problems that involve processing high volumes of data and intelligent responses. Thanks to its data-driven approach, DNNs offer designers a relatively easy framework that involves model construction through interaction with data. Additionally, DNNs are now state of the art in applications that involve Computer Vision (CV) and Natural Language Processing (NLP). So, developers can resort to these well-established models to classify pictures, track objects, enhance image quality, process written language, generate text, etc, either as a main or auxiliary application. However, implementing DNNs in resource-constrained devices and latency-sensitive scenarios can be challenging, especially when these models are constructed with multiple layers and parameters. This master's thesis discusses these problems, and presents two alternative Neural Network (NN) architectures that aim to ease these deployment issues: Asymmetric Autoencoders (AAEs), and Early-Exit DNNs (EE-DDNs) for semantic segmentation. The former offers a resource-constrained friendly architecture to enable the implementation of autoencoder-based solutions in paradigms like Internet of Things (IoT), whereas the latter has the objective to expand EE-DNNs success with image classification to semantic segmentation, addressing some implementation issues that come with this problem transition. Overall, the results show the importance of incorporating architectural changes to DNNs that hopefully will fuel future research on efficient and lightweight DNNs for resource-constrained and latency-sensitive applications.

## ACRONYMS

## MATHEMATICAL NOTATION

*The notation compiled below represents the established standards adhered to by this master's thesis. Any deviation or introduction of notations inconsistent with these standards will be explicitly addressed in the text when deemed necessary. Notations listed with more than one meaning are clarified when they appear in the text.*

$x$ - a unit of input data (e.g. sample, or entry value )

$y$ - a unit of ground-trough data

$\hat{y}$ - a unit of output data (e.g. generated output, inference)

$X$ - input data or the input data domain

$Y$ - ground-trough data or the output data domain

$\hat{Y}$ - output data

$\sigma$ - activation function

$W$ - weight matrix

$b$ - bias vector

$b_i$ - $i$-th early exit (or side branch)

$lr$ - learning rate

$i, j, k$ - indices

$s$ - stride

$r$ - dilation rate

$C$ - a collection of classes on a given problem

$TP_c$ - True positives of the $c$-th class

$FP_c$ - False positives of the $c$-th class

$FN_c$ - False negatives of the $c$-th class

$H[X]$ - Entropy of random variable $X$

$H[X|Y]$ - Entropy of random variable $X$ conditioned on $Y$

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the last decade, Deep Neural Networks (DNNs) has become a relevant and popular tool to handle a variety of problems that involve processing high volumes of data and intelligent responses. For instance, recent state-of-the-art applications in Computer Vision (CV) and Natural Language Processing (NLP) employ DNN [1, 2]. DNNs can be designed to classify pictures, track objects, enhance image quality, process written language, generate text, etc. However, implementing DNNs in resource-constrained devices and latency-sensitive scenarios can be challenging, especially when these models are constructed with multiple parameters. DNNs can easily require a lot of storage space and computation, which can lead to significant power consumption and processing time. Moreover, the size of processed data can worsen these issues, whether by its dimension (*e.g.* image size) or the rate at which they are generated. To address these problems, many researchers are focused on designing alternatives to construct computationally-efficiently DNNs [3, 4]. This work investigates alternative DNN's designs that focus on reducing the number of parameters that are used by the model. A carefully designed DNN can deliver comparable results with overparameterized models and can handle simpler data with fewer computations, thus saving time and resources.

## 1.1 Why resort to DNNs?

Usually, complex problems demand complex solutions to deal with them. Problems may require non-linear models, which can be difficult to devise, or dealing with challenging data. They can require specialized expertise that might not be easily available, or involve a substantial time investment in data interpretation and problem modelling. Moreover, if such requirements can be met, some solutions cannot generalize well if too many restrictions are adopted. Taking advantage of the recent improvement in hardware technology and data availability, Deep Learning (DL) became an important framework that offers powerful and easy-to-implement solu-

tions for many of these problems [5]. DL enables a data-driven approach in which a functioning model can be obtained autonomously through its interaction with data. During the processing of tuning, referred to as learning, a model adjusts its parameters in a manner that allows it to extract and combine data features to perform the intended task. Hence, when given enough data and time, DL models are expected to learn how to adequately perform an intended task, obtaining the capacity to identify data particularities needed for better generalization.

Central to DL are DNNs. DNNs are layered models that enable hierarchical aggregation of information, where data that flow from the input to the output layers traverse intermediate layers that extract and fuse its features. These models provide a relatively easy framework in which performance improvements can be achieved by inserting more intermediate layers or adding more computational resources to existing layers. The premise is that a model with more resources can extract more distinct and complex features. With additional cascading layers, the last layers can merge primitive features extracted in layers that precede them, generating more elaborate data and enabling the DNN to become a more complex model. For instance, in CV, it is a common practice to construct layers with multiple filters that extract distinct features in parallel and to add more layers to enable the DNN to extract features in multiple resolutions [6].

In a future where large amounts of diverse data are expected to be ubiquitous, DNN tends to become an increasingly relevant tool. For instance, vision-aided wireless communication is an emerging paradigm that promises to assist communication applications with visual data by exploiting the increased availability of cameras in urban environments [7]. Additionally, with sensing equipment constantly becoming cheaper and new devices being integrated into the home and urban environments, new applications must be capable of handling heterogeneous data [8, 9]. Hence, understanding DNN limitations and learning how to address them will be fundamental to using this model effectively.

## 1.2 When "Deep" becomes an issue

Even though adding more layers to a DNN can lead to performance improvements, this addition can make implementing DNN-based applications in resource-constrained devices and latency-sensitive scenarios difficult. More layers mean more parameters to store, which can be an issue when embedding DNNs in devices with low memory. Additionally, the computation needed to reach the DNN's exit in models with multiple layers can lead to significant processing time, which is problematic in applications that require fast responses. State-of-the-art DNNs are usually implemented in high-performance equipment, thus deploying these models in non-cutting-

edge devices can increase response delay. Moreover, deep DNNs have significantly more parameters to fit, requiring more data to adjust their parameters. With more parameters, a model is more prone to adjusting perfectly to data presented in the learning process, harming its generalization capabilities. Thus, constructing DNNs with multiple layers in applications with limited available data may not be advisable, meaning that performance improvements must be sought through other approaches.

Usually, developers tend to stick to a similar approach when designing DNNs. DNN innovation is usually focused on discovering new ways to make models with multiple layers learn effectively, or through the proposal of new and more powerful layers (to replace the ones currently in use). Even though both approaches help advance the performance of DL models in general, they are not accessible to a wider audience that is not very familiar with the theory behind DNNs. Moreover, designers might not be interested in cutting-edge DNNs if they can't meet their application's restrictions. For example, people who want to employ DNNs to support applications in CV-assisted wireless communication may not be interested in a state-of-the-art DNN capable of identifying dozens of classes if it takes a lot of time to deliver its answer. Additionally, those interested in deploying DNNs in Internet of Things (IoT) and edge computing may prefer compact and energy-efficient models to enable easier implementation on resource-constrained devices. Obviously, these developers can benefit from the advances in the DL community, but they will likely be satisfied if they can design a functional DNN that meets their applications' requirements with the current tools available. This work addresses these concerns by presenting alternative DNN architectures for resource-constrained devices and latency-sensitive applications. The new architectures exploit DNN's versatility to deliver models that work well with fewer layers and parameters to enable wider adoption of DNN-based solutions in areas such as IoT, edge computing, CV-assisted wireless communication, and autonomous driving.

## 1.3 Work contributions

This work proposes two alternative DNN architectures. The first is the Asymmetric Autoencoder (AAE), an alternative to the traditional symmetric Autoencoder (AE) for resource-constrained devices. AEs are a common choice when dealing with dimensionality reduction and data enhancement problems. They are suited for applications that require the extraction of data's defining features to recover the information inserted at the AE's input. Its architecture is divided into an encoder-decoder model, where the first layers are responsible for extracting the features needed for "summarizing" the input data in lower dimensions, whilst the remaining layers are responsible for reconstructing the original data. The traditional

approach is constructing AEs where the encoder mirrors the decoder's layer disposition. This can be problematic when embedding the encoder on devices with limited space and energy concerns. For example, many AE-based solutions that deal with IoT data have been proposed recently in problems that involve data compression and noise removal [8, 9]. Data transmission is a power-demanding application in IoT networks, meaning that compression systems to reduce redundant data transmission are always in high demand. Additionally, data enhancement applications can assist in many problems, from noise suppression to fusing heterogeneous data, to generate more refined information. However, leaning on the traditional approach of inserting more layers to seek performance improvements can be problematic in these scenarios because IoT networks are usually composed of resource-constrained devices. The proposed AAE addresses this issue by presenting an asymmetric architecture, where the number of layers and other resources in the encoder is smaller than in the decoder. The results show that AAEs are a suitable alternative to the symmetric AEs, even capable of outperforming their symmetric counterparts in some cases where there are few training examples.

The second proposal is Early-Exit DNNs (EE-DNNs) for semantic segmentation, a CV problem that involves pixel-level classification. Semantic segmentation focuses on identifying different elements within a picture, dividing an image into regions and classifying them accordingly. Applications like autonomous driving, smart health and vision-aided wireless communication are interested in semantic segmentation [1, 7] because it is fundamental for identifying obstacles and sensitive objects in an image. DNNs are very successful in semantic segmentation, with many models being state-of-the-art in many CV datasets [1, 10, 11]. However, these cutting-edge models are characterized by having multiple layers, which can be an issue when implementing these DNNs in constrained hardware or when working with applications that require low latency. Following the recent success of EE-DNNs in image classification [12, 13], this work shows that the proposed DNN can deliver segmented images that can be useful in applications that demand fast response. Additionally, the proposed architecture can be easily split to allow some layers to be placed closer to the end users in local devices and edge instances. In this approach, data is forwarded to remote cloud servers only when necessary, meaning that inference can happen closer to the end device whenever possible, minimizing response delay and power expenditure associated with data transmission.

In summary, the main contributions of this work are:

- proposing Asymmetric Autoencoder for dimensionality reduction and data enhancement applications in scenarios that involve resource-constrained devices;
- extending EE-DNNs for semantic segmentation, enabling this CV application

in edge-cloud co-inference implementations that offer a resource- and latency-efficient deployment; and

- promoting a different approach for DNN design that inserts architecture-based restrictions to construct models more suited for resource-constrained and latency-sensitive applications.

Additionally, the papers that are by-products of this Master's thesis can be found in Appendix A.

## 1.4 Thesis outline

The remainder of this thesis is organized as follows. First, chapter 2 presents proposals that employ DNN in IoT networks and EE-DNNs in image classification. Also, other approaches to ease DNN implementation in resource-constrained devices are quickly revised in this section. Chapter 3 brings a summarized review of DNN, presenting the theory needed to understand the topics of this thesis, as well as a discussion on what kind of problems DNNs are well-suited and obstacles to its implementation that this work pretends to address. Next, chapter 4 presents the AAEs, experiments showing their advantage to the traditional AEs, and remarks on future projects where the proposed model can be employed. After that, chapter 5 discusses the usage of EE-DNNs in semantic segmentation, outlining proposals on how these models can be adopted effectively in time-sensitive applications and edge-cloud co-inference scenarios. Lastly, chapter 6 concludes the paper, discussing the results in a broader view and drawing future directions.

# Chapter 2

# DNN Research and Related Work

The main focus of this thesis is to provide an analysis of DNNs implementation in scenarios where it may face resource- and time-constraints. Multilayered Neural Networks (NNs) are fundamental to many applications, as they can extract and combine complex features to enable more elaborate and refined inferences. However, these deep models, together with data dimensionality, may require storage and computational resources that can be scarce or non-existent in cheaper and smaller devices. For example, to keep low implementation costs, many IoT networks tend to employ resource-constrained devices as sensor and actuator nodes, in which implementing a DNN can be challenging. Additionally, processing time is a concern in time-sensitive applications. Usually, DNNs are designed using cutting-edge equipment, thus implementing them in other devices may impact inference time negatively and can vary a lot from equipment to equipment. Moreover, when dealing with resource-constrained devices, it is common to choose to store these models in remote servers (*e.g.* cloud) rather than implementing them locally, which adds transmission delays. Although we can expect hardware improvements that can ease some implementation concerns for some of the current state-of-the-art models, we can expect that the discussed issues with local implementations will not disappear, as the continuous development of NN theory will likely bring bigger and more complex DNNs. Hence, having alternative NN architectures specially designed for resource-constrained and latency-sensitive implementations can an important tool in the arsenal of any developer.

A quick literature review is provided in the remainder of this chapter. The works are divided according to the two main themes discussed in this thesis. First, we start with projects that resort to DNNs in IoT applications, and are followed by works that discuss multi-exit DNNs for distributed implementations. These works are presented with commentary on how increasing DNN size can be problematic in these cases. Finally, in the end, a review of other works that try to address DNN implementation issues is presented. These follow a different approach from

this thesis's aim and can be complementary.

## 2.1 DNNs in IoT

IoT data is characterized as being extremely heterogeneous [8], which can make processing data and developing applications to deal with it challenging. A single IoT network can comprise distinct devices, each with its own sampling rate and utilizing hardware with varying quality. A single network or application may collect and transfer distinct phenomena, and environmental noise can add another degree of difficulty. Additionally, another issue that arises from IoT network size and operation is the amount of collected data and the need to transfer them among nodes. In many cases, especially networks that involve sensing applications, employing multiple sensor nodes and requiring high sampling rates is needed to monitor the environment appropriately. Efforts to minimize data size are in high demand to assist in saving transmission bandwidth [14]. Many researchers are resorting to DNNs to deal with IoT data [9], as these models can offer a data-driven application design. With a DNN, developers can use historical data to make the model learn how to deal with the intrinsic difficulties of the desired application, thus enable intelligent applications in which sensors and actuators can adapt to environmental changes or make automated decisions.

Usually, to keep implementations cost low, a typical IoT network is composed of low-cost resource-constrained devices. Thus, DNN size plays a crucial role when designing an NN-based application for this type of network. Specifically, designers must take into account the amount of resources these models will impose on network nodes, both in terms of memory space and energy consumption. Particularly, the traditional approach of performance improvements through increased NN depth should be rethought. In particular, AAEs (proposed in chapter 4) aim to address this problem when designing applications that employ AE.

### AEs in IoT

Given the IoT data characteristics discussed previously, many applications require some sort of dimensionality reduction or data enhancement. For instance, data compression helps save bandwidth and transmission resources, noise removal can improve data quality, and extracting main data features from high volumes of data is fundamental to give a better understanding of a monitored phenomenon. Usually, AEs are the models of choice when dealing with dimensionality reduction and data enhancement problems. They are an NN with an encoder-decoder design constructed to reconstruct inserted data (see chapter 3 for more details). Hence,

several works that involve data compression and noise suppression in IoT networks use AEs [9]. Alsheik et al. [15] demonstrate how a shallow AE-based compression schemes can deal with IoT data. The authors design a shallow AE, which has a single intermediary layer, to handle temporal series. With it, they showed that this model can deliver a lightweight compression scheme that significantly reduces energy consumption. As expected, increasing the depth of this symmetric AE yields better data compression results. As discussed previously, the more layers an NN, the more complex functions it learns. For instance, Ghosh et al. [16] constructs a dimensionality reduction application to decrease data size for transmission, in which AEs with more layers outperformed the ones with fewer layers. However, the layer disposition of the encoder in a typical AE mirrors the decoder's, thus increasing the number of layers can be a problem. More encoding layers incur storing more parameters and running more computations locally in the sensor. The proposed AAE offers an alternative IoT-friendly design whose aim is to reduce the number of layers stored in sensing nodes. In particular, in the experiments in chapter 4, the number of encoding layers in all AAEs is the same as in Alsheik et al.Thus, these chapter's results show that it is possible to keep the lightweight implementation benefits shown in that work without affecting performance.

Having applications that can deal with environmental noise is necessary for many IoT applications, and many researchers have resorted to AEs to remove this unwanted feature. In this direction, two recent works use Denoising Autoencoder (DAE) to remove noise from data. Laakom et al. [17] propose a new loss function to help the model learn how to remove the noise. Even though they consider data compression, they focus on denoising, showing that autoencoders can deal with both challenges. Additionally, their model is symmetrical and has multiple encoding layers, as it does not consider IoT data. In another work, Lee et al. [18] propose a modification to DAE, which shifts from the traditional approach of learning how to reconstruct the noiseless signal to extract the noise, subtracting it from the signal. They show that their approach outperforms DAE in the evaluated scenario, but differs from ours by resorting to a symmetrical architecture and not addressing data compression. Overall, AAE can offer a compression scheme that can deal with environmental noise with fewer encoding layers, as we incorporate limited encoder size as a learning regularizer.

An asymmetric AE design was previously analysed [19], but adopting an encoder containing more layers than the decoder. This architecture is the opposite required in typical IoT scenarios, as the bulk of computation would be shifted towards the resource-constrained nodes. Ideally, in an IoT network, we want applications to outsource most computation to a resource-rich central node. This can be addressed by the proposed AAE. Hence, the key contribution is presenting asymmetrical NN ar-

chitectures that can be a valuable alternative when implementing applications that use AEs in IoT networks or other implementations with similar implementation issues. Through chapter 4 experiments, we will see that AAEs can deliver comparable performances to their symmetrical counterparts, capable of even outperforming them. Hence, in addition to its architecture suited for resource-constrained devices, AAEs can be fundamental to NN deployment in IoT. Table 2.1 summarizes the differences between chapter 4 work and other proposals that involve data processing with autoencoders.

Table 2.1: Summary of related work resorting to autoencoders for data compression (or dimensionality reduction) and noise removal.

| Reference | Data Compression & Dim. Reduction | Noise Removal | Single Encoding Layer | Asymmetric Design |
|---|---|---|---|---|
| [15] | ✗ | | ✗ | |
| [16] | ✗ | | *partial* | |
| [17] | *partial* | ✗ | | |
| [18] | | ✗ | | |
| [19] | ✗ | | | ✗ |
| chpt. 4 | ✗ | ✗ | ✗ | ✗ |

## 2.2 Ubiquitousness of CV applications

DNNs have been used in a variety of problems. Among them, one where they are very successful are CV problems, where DNNs are now state-of-the-art [1]. This success has prompted many researchers and developers to include DNN-based CV models in autonomous driving, smart heath, and environment monitoring [1], either as the main or auxiliary applications. In the latter case particularly, with cameras becoming cheaper and more ubiquitous in urban and building environments, many applications and new paradigms are seeking to leverage this visual data abundance to support existing functionalities. For instance, vision-aided wireless communication is an emerging paradigm that aims to combine CV applications to communication systems and protocols [7].

All this interest in CV applications, DNN-based solutions in particular, means that the environment where these models will be implemented is extremely heterogeneous. Additionally, this diversity also means different performance standards. For instance, time-sensitive require fast responses, and energy-constrained require models to be less resource-demanding. Another issue with DNN implementation, and relevant to the previous examples, is NN size. Models with multiple layers demand more computational resources and take more time to finish computations.

And although cloud-based implementations are an option for resource-constrained implementation, sending data to the cloud can also add significant delay and energy consumption. EE-DNN and DNN partitioning combined give an alternative where portions of a DNN can be distributed and placed in local devices, edge, and cloud. This setup is successful with image classification, where simpler image data can be labelled faster with an auxiliary output. As will be shown in chapter 5, this concept is extended to semantic segmentation, showing how this setup can address the difficulties of other CV problems to realize a resource-efficient and faster semantic segmentation.

## Early-exit DNNs and DNN partitioning

Teerapittayanon et al. [12] proposed BranchyNet as a first multi-output DNNs for faster image classification. Their work showed how these multi-exit models were faster on average than the unbranched model they were based on, laying the foundation for most of EE-DNN research. These models can be dismantled and placed in local devices, edge and cloud instances, offering a hierarchical inference system [20]. Simpler data can be classified in earlier stages, which can consist of a small set of layers connected to an early exit only hard treat data are offloaded to a cloud server, which has the resource-demanding remainder of the DNN. This opened the opportunity for fast and lightweight DNN implementations, like the one proposed by Lo et al. [21] that adopts a DNN with two exits, creating a lightweight DNN that is placed in the edge and another that is placed at a remote server (*e.g.* cloud). Pacheco and Couto [22] combine EE-DNNs with DNN partitioning to offer an edge-cloud co-inference scenario. In the proposed partitioning method, layers connected to early layers are positioned on the edge to enable first classification attempts to occur close to end devices, and the remaining layers are placed on the cloud. Later, Pacheco et al. [13] investigated models with different numbers of early exits and investigated different exit thresholds [13]. In both cases, they showed the advantages of EE-DNNs in minimizing inference time, as classifications in early exits can save the time-demanding offloading process. However, all these works tackle image classification. Usually, inserting early exits in a Convolutional Neural Network (CNN), which are the base of most DNNs that deal with CV, can be hard due to the large dimensionality of image data, especially in layers close to the DNN input. To address this, EE-DNNs for image classification can adopt hard downsampling and pooling methods to decrease data size [20]. However, semantic segmentation doesn't work well with these methods because they can destroy localized and structured information extracted in early layers. Additionally, most exit criteria rely on classification probabilities, comparing them with a pre-defined threshold to determine if data

should be classified in an early exit. This can be infeasible in semantic segmentation because it involves pixel-level classification. The proposed EE-DNN for semantic segmentation addresses these issues. Table 2.2 summarizes the differences between chapter 5 work and other proposals that use EE-DNNs.

Table 2.2: Summary of related work resorting to EE-DNNs and that also resort to DNN partitioning.

| Reference | Image Classification | Semmantic Segmentation | DNN Partitioning | Entropy Based Exit criterion |
|---|---|---|---|---|
| [12] | ✗ | | | |
| [21] | ✗ | | *partial* | |
| [22] | ✗ | | ✗ | |
| [13] | ✗ | | ✗ | |
| chpt. 5 | | ✗ | ✗ | ✗ |

## 2.3 Other alternatives to ease DNN implementation

With the popularization of Machine Learning (ML) in recent years, especially thanks to the recent success of DNNs, a great effort is being spent in simplifying model requirements to embed them in resource-constrained devices. Tiny Machine Learning (TinyML) is a paradigm that emerged recently, focusing on designing ML models for low-power and low-cost microcontrollers [3]. Among different approaches, TinyML usually relies on parameter pruning and quantization to offer more compact and computationally efficient models [3, 4]. Although crucial for implementing DNNs in resource-constrained devices, quantization can lead to performance degradations because 32-bit floating point parameters are mapped into representations that require fewer bits, thus losing representation precision. Additionally, the current best quantization techniques can offer a maximum size reduction of four times [4], and pruning-like techniques are more effective in over-parameterized models that have many infinitesimal parameters. This means that DNN that are big to begin with naturally take more space than models that are designed with fewer layers and parameters. The proposed architectural changes use an unrelated approach that can help TinyML. For instance, an IoT sensing node is expected to collect data from multiple sources [23], so it will likely store multiple ML models to treat each one individually or will require a complex model capable of dealing with this heterogeneous data. With the proposed AAE, one can start with a more compact model, which can downplay the need for parameter quantization and pruning. Additionally, when deploying an EE-DNN with DNN partitioning, we can limit the number of layers to be stored in edge- and end-used devices, meaning that softer quantization and pruning techniques can be adopted.

Another paradigm that aims to reduce DNN size for implementation is knowledge distillation [24], which focuses on replacing a DNN with multiple layers with one more compacter. Using the complex model as a reference, the idea is to extract the most essential learned transformation, capable of replacing a big set of layers with one or a smaller set of layers. This can be done by training the two models together, trying to make the smaller model emulate the data generated by the bigger one, and other approaches that are continuously being proposed [24]. Again, architectural changes, such as the ones proposed in this thesis, and knowledge distillation can be mutually beneficial. For instance, AAEs can encoders to replace a big set of DNN layers, and knowledge distillation can help train and design EE-DNNs.

# Chapter 3

# DNNs and its Applications

The popularization of solutions and applications that resort to Machine Learnings (MLs), and in particular Deep Learnings (DLs), has seen Deep Neural Networks (DNNs) become an indispensable tool in the arsenal of developers and researchers that deal with high volumes of data and that require a data-driven approach. DNNs are now state-of-the-art in many problems in Computer Vision (CV) and Natural Language Processing (NLP), and are especially superior to design approaches that involve trial and error and expertise knowledge. For instance, the traditional approach in Computer Vision (CV) is selecting feature extractors manually, a process that DNNs can automate by interacting with data [25]. Additionally, DNNs offer a relatively easy framework in which performance improvements can be sought through an increase in model's complexity via the addition of more layers and more specialized computational units. However, implementing DNNs with multiple layers can be difficult in scenarios that involve resource-constrained devices and that require low-latency responses. In this chapter, we provide an overview of DNN theory, covering its basics and components used in subsequent experiments. We conclude by discussing challenges in constructing DNNs with multiple layers and computing units.

## 3.1 Machine Learning basics

ML is the collection of data-driven models and algorithms that adjust their parameters by interacting with data in a process commonly referred to as "learning", which requires little or no human intervention. ML models can determine their performance in the task for which they were designed with the help of a cost function that gives them a manner in which they can track their progress and determine which parameters need to be adjusted. Ideally, through consecutive interactions with task-representative data, the ML model parameters will be adjusted in a manner that they capture the task particularities, allowing the model to be effective and generalize

well.

The usual ML approach consists of two steps. In the first step, which is referred to as training, the ML model interacts with a large quantity of task-representative data to adjust its parameters. "Learning" effectively happens during this process, as it's during training that a model captures knowledge about the task for which it was designed. After training is complete, a final evaluation step is executed to measure the trained model's generalization capabilities. In this step, referred to as testing, the model is presented with data not present in the training process, and a final score is obtained to estimate its field performance.

## 3.2 What are DNNs and what are they good for

Basically, a NN is an ML model that implements a mapping from a data domain ($X$) to an "answers" domain ($Y$), which has the desired outputs of a given task. For instance, the data domain might contain observations of a phenomenon of interest or all possible images of a specific size, and we want a model that relates each element in $X$ to the appropriate classification or action in $Y$. Ideally, there is a perfect function $f : X \to Y$ that correctly associates every possible input to its appropriate output, and we want the designed NN to represent it as closely as possible. How close to $f$ we can reach will depend on how many parameters the NN has, if the model is constructed adequately, and the quality and quantity of data used in the learning process. The first two requirements are related to the model's architecture and restrict the complexity of functions that the NN can represent. The latter is fundamental to the success of NN's pursuit of $f$ because a model will only learn if given good examples, even if constructed correctly.

The fundamental component of an NN is a computational unit (commonly referred to as neuron) that combines data and applies a non-linear function (referred to as activation function) to it. Neurons can be grouped to form a structure known as layer that receives the same data instances as input, allowing for the extraction of distinct and complementing features in the same data. Additionally, layers can be linked to enable the merging and fusion of extracted information into more complex features. Usually, a neuron combines the input data using a weighted sum and feeds the result to the non-linear function. Thus, a layer can be thought of as an affine transformation followed by a non-linear transformation. If we denote $\sigma$ as the non-linear function, the output can be represented as

$$y = \sigma(W \cdot x + b), \tag{3.1}$$

where $x$ and $y$ are the input and output data, respectively, $W$ is the weight matrix

whose entries represent the weights of the layer's neurons, and $b$ is a bias vector. The training process objective is to learn the adequate values of $W$ and $b$ so that the resulting NN can perform the application for which it was designed. For example, suppose that $y$ represents a class encoding in an image classification problem where each neuron produces an encoding entry. Then we want the learned values of $W$ and $b$ to effectively combine the features received by this layer to produce the correct outputs.

DNNs are an NN variation characterized by having many layers and adjustable parameters. DNNs can use this big number of layers to produce complex information, exploiting the consecutive application of non-linear transformations to combine primitive information into more refined data. Most layers follow equation 3.1, thus by linking their inputs and outputs we can construct a composite function that leverages the chaining of equation-3.1-like functions to construct a more complex non-linear model. This can increase the number of functions that a DNN can emulate significantly. Both these attributes are particularly useful when dealing with complex problems and are the reasons why DNNs with multiple layers can be powerful ML models. However, with too much freedom comes the risk of learning useless functions because these deep models are more difficult to train and require significant training data. This is one of the reasons why, even though NN is an old concept, the popularization of DNNs occurred in the last decade after advances in hardware and data availability that enabled the efficient construction and training of models with multiple layers and parameters [5].

DNNs excel excel at problems that require extraction of information from complex data and in which there is a clear relation between input data and the desired output, even when this relationship is hard to determine. Additionally, DNNs can work with raw or minimally-processed data because they can learn the transformations necessary for extracting required features [5]. For example, in image classification, the choice of feature extractors depends on the specificities of the application in which they will be employed, which can be a demanding task if done manually and usually requires expertise knowledge [25]. DNNs can learn the extractors they need on the fly while learning how to pair each image with its correct label. Thus, DNN can be seen as a useful tool in hard-to-model problems or when dealing with large amounts of heterogeneous data, where processing data can be challenging. Obviously, preprocessing data can be helpful to DNNs, but these models can dismiss data treatment at the expense of more layers and training data.

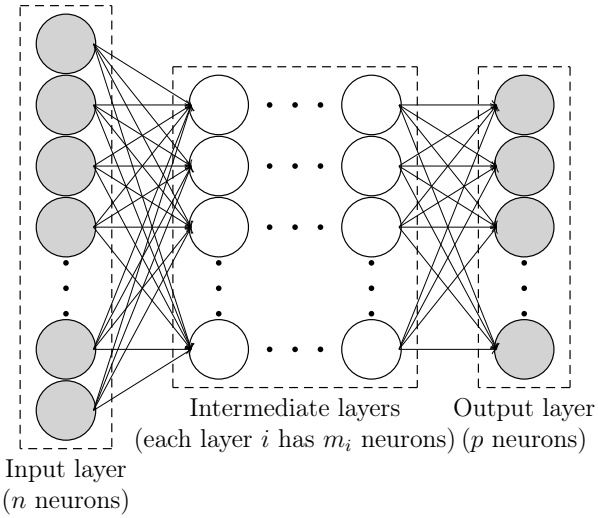



Figure 3.1: A fully-connected-feedforward DNN. In this implementation, data flows from the input to the output layers without cycles, and a neuron connects with all neurons of the following layer.

## 3.3 Different models for different problems

The simplest DNN implementation consists of a model where neurons are placed in layers, with a neuron connecting with all neurons placed in the next layer, as shown in Figure 3.1. Because of this feature, this DNN can be referred to as fully connected, given that all neurons in adjacent layers are connected with one another. Additionally, in this implementation, data flows through the layers without cycles, thus this model can also be referred to as feedforward. Feedforward DNN is the most common NN implementation because of its architectural simplicity and versatility. Although a simple and effective implementation, a DNN like the one in Figure 3.1 can be enhanced with the addition of different layers that are more suited for the data being processed. And the number of neurons per layer can vary from application to application, to meet implementation and performance requirements. In the following subsections, we will review a few DNN variations and layers used in the experiments of this thesis.

### 3.3.1 Autoencoders

Autoencoder (AE) is the NN most used in problems that involve dimensionality reduction and data enhancement, such as data compression and noise suppression. The AE is constructed in a manner that allows it to produce as output data inserted in its input layer, functioning as a $f : X \to X$ mapping. The objective of this NN implementation is to learn how to extract information from data so that the NN's output is a replica or an improved version of the original input. In dimensionality reduction, an AE should be capable of generating smaller representations of the

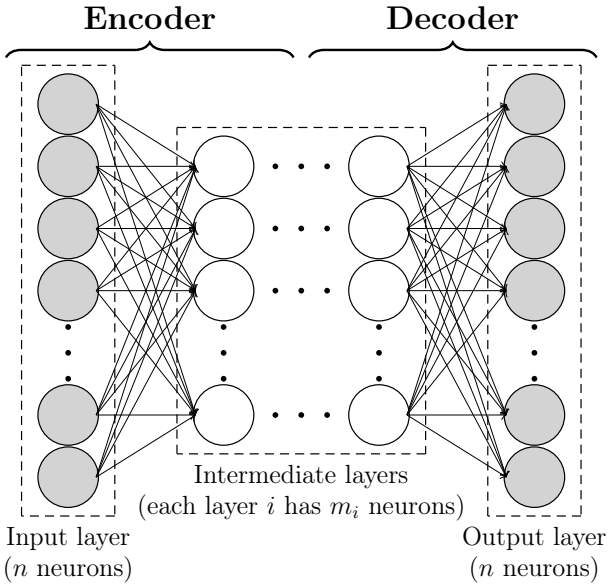



Figure 3.2: A typical AE. Its prominent feature is a similar neuron configuration in the input and output layers, which enables the recovery of data inserted in the model's input.

inserted data, and to ideally function as an identity mapping. In data enhancing problems like noise removal, an AE must learn to extract defining features of the input data so it can generate an output without unwanted elements. Hence, it's a common practice to construct these NN with input and output layers that have the same number of neurons, such as seen in Figure 3.2.

The AE's architecture can be divided into an encoder-decoder model, where the first layers extract features from the inserted data and the remaining layers reconstruct the original data using these features extracted in the encoding layers. Hence, we can interpret the encoding layers functioning as a $f : X \to F$ mapping and the decoding layers as $g : F \to X$, where $F$ is the feature domain. For example, in a dimensionality reduction application, $X$ can be where the compressed representations of elements from $F$ reside. Thus, in this interpretation, the whole AE can be thought of as $g \circ f$ rather than a single function, where $f$ is a feature extractor and $g$ is a function that produces the original data from extracted information.

A common approach in AE design is constructing it with intermediate layers that have fewer neurons than the input and output layers, much like the configuration shown in Figure 3.2. Sometimes referred to as undercomplete AE [26], this implementation creates a bottleneck that forces intermediate data to be smaller than the input. Hence, the AE architecture becomes a restriction in the learning process that forces the NN to learn compressed representations. In the remainder of this thesis, undercomplete AEs will be referred to as AEs, which is usually the case.

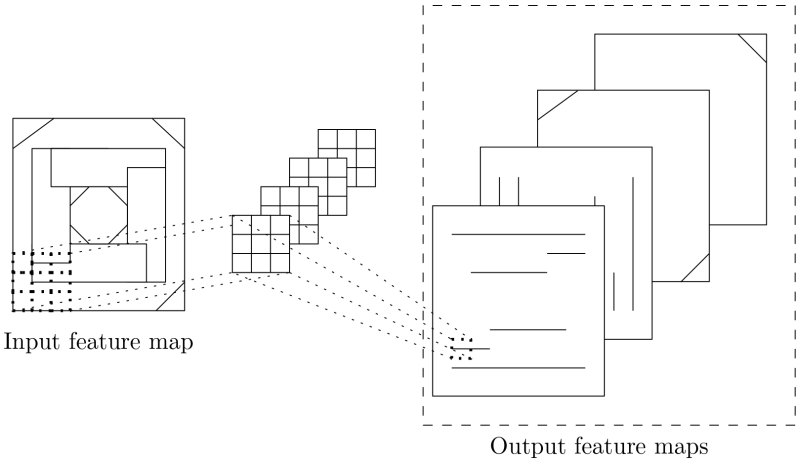
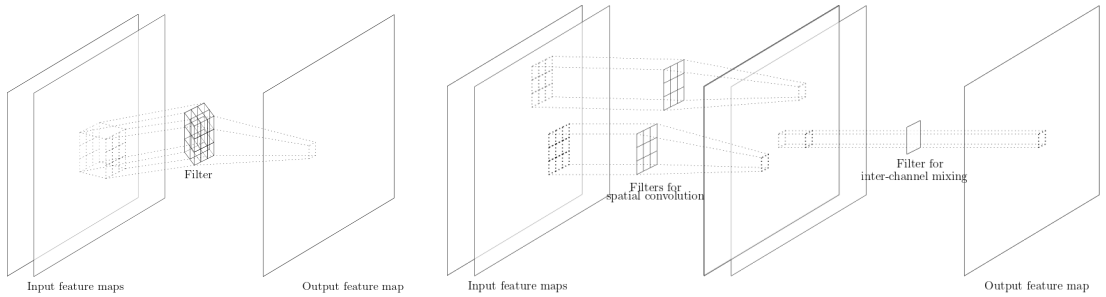



Figure 3.3: Convolutional layer's typical operation. As we stack convolutional layers, the deeper DNN layers can extract complex features by combining and extracting new information from output feature maps of previous layers.

### 3.3.2 Convolutional layers

Even though fully-connected DNNs can learn a large number of functions if constructed with enough neurons [27] and layers, computational units more suited to exploit data particularities can lead to more powerful models and that require fewer parameters. This is particularly useful in deeper implementations, which naturally have a lot of parameters, because we can significantly reduce the number of neurons and simplify the training process. In this direction, convolutional layers are popular in problems requiring processing data with strong relations between neighbouring entries [28], such as images and signals with temporal correlations. Instead of having neurons that connect with all neurons of the next layers, convolutional layers employ filters (also referred to as kernels) constructed with few parameters and that extract features from data in a sliding-window fashion. Operationally, these kernels function as a small set of neurons that are replicated along an NN layer, limiting the number of neuron connections [28]. Furthermore, these filters extract similar features in different portions of data, which is useful for problems like CV.

A common approach is constructing convolutional layers with multiple filters to extract distinct and complementing features. For example, suppose we have a convolutional layer that extracts horizontal, vertical, and diagonal lines tilted to the right and left, using four different filters. Then, if given the picture on the left of Figure 3.3, the convolutional layer will produce the pictures on the right as output. Later, if we add more convolutional layers after this layer, these features can be combined to generate more complex data. To distinguish the convolutional layer's output from the datum generated by each filter, the latter is usually referred to as a feature map. Additionally, it is common to say that each feature map enters (and exits) a convolutional layer through an input (output) channel. These nomenclature




(a) Regular convolution. (b) Depthwise-separable convolution.

Figure 3.4: Comparison between regular convolutional layer and the one that uses depthwise-separable convolution.

conventions will be adopted henceforward.

Even among convolutional layers, there is a diversity of implementations to address different problems in feature extraction and DNN training. When a convolutional layer receives multiple input feature maps, it is expected to effectively integrate inter-channel information to generate new and more complex features. In a typical implementation, filters are required to learn how to perform spatial and cross-channel correlations simultaneously. Depthwise-separable-convolutional layer is an alternative implementation in which feature extraction and inter-channel mixing operations are performed separately. This layer is shown empirically as being capable of learning rich representations with fewer parameters than the typical implementation [29]. This implementation leans on the assumption that the whole convolutional layer operation becomes simpler and more efficient if these operations are split. Thus, employing layers using depthwise-separable convolution can be useful in problems with a shortage of training samples, like the ones studied in chapter 4. Figure 3.4 brings a comparison of the two convolutional layers, in which both receive two input feature maps and produce a single output feature map. Notice that, in Figure 3.4(b), two sets of convolutional layers (two spatial and one inter-channel) do the work of a single filter in Figure 3.4(a). Hence, in the depthwise-separable case, each filter can specialize in a portion of the convolution task, simplifying the training process.

Atrous-convolutional layer [10], which is also referred to as dilated-convolutional [30], is an implementation that employs filters that perform atrous convolution to extract low-resolution features without data decimation. DNNs constructed with regular convolutional layers require consecutive data downsampling in order to extract low-resolution features. This is an issue with CV applications like semantic segmentation that need localized information [31]. Feature maps of layers closer to the DNN's exit, which come after a sequence of downsampling steps, no longer have access to localized information extracted in layers that dealt with data

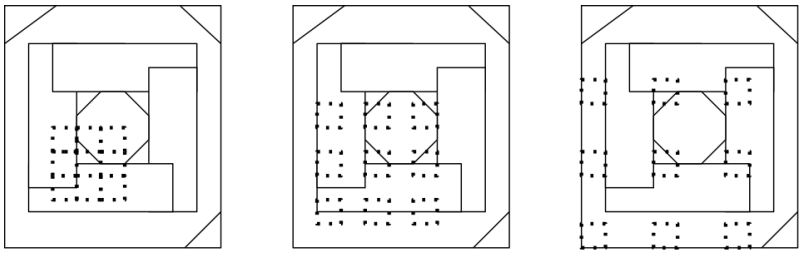



Figure 3.5: An 2D atrous convolution, for $r$ equal to 1, 2 and 3. When $r = 1$, it functions as a regular convolution. Overall, it is noticeable that with bigger $r$, the greater is the filter's FoV.

in higher resolutions. Atrous convolution enables the extraction of low-resolution features in higher resolutions because it increases a filter's Field of View (FoV) by skipping neighbouring samples in a datum. For each entry $i$ in an $y$ output feature map

$$y[i] = \sum_k x[i + rk]w[k], \tag{3.2}$$

where $x$ and $w$ are the input feature map and convolutional filter, respectively, and $r$ is the input sampling rate [10]. In practice, an atrous-convolutional layer functions much like a regular convolutional layer if we replace $w$ with a bigger filter that has $r - 1$ zeros between two consecutive values. Hence, an atrous-convolutional filter FoV is increased because it utilises values that are multiple samples apart, as shown in Figure 3.5, enabling it to extract low-resolution features that are dispersed over a wider range of samples in higher dimensions. However, it is known that employing a single atrous rate can be harmful to the overall DNN, as this can lead to loss of information [11]. Given its inherent characteristic, a large portion of high-resolution information is lost in this convolutional process. Hence, it is common (and advisable) to construct layers that utilise different atrous rates, such as Hybrid Dilated Convolution (HDC) and Atrous Spatial Pyramid Pooling (ASPP) layers. With these layers, an NN can extract features in multiple resolutions from the same data instance, exploiting the advantages of atrous convolution to the fullest.

An issue that may occur when constructing a DNN with multiple convolutional layers is that it can make training them significantly harder, leading to bad models. An ingenious solution adopted in many state-of-the-art DNNs to add feature maps extracted in early layers to the ones generated later, thus propagating features extracted earlier further down the NN graph. Referred to as residual connections [32], this modification is the building block of Residual Network (ResNet), itself the backbone of many state-of-the-art DNNs in CV applications. Formally, supposing that $f$ represents the set of layers bypassed by the residual connection and $h$ as the set

of layers without this modification, the learning problem becomes

$$y = h(x) = f(x) + x, \tag{3.3}$$

For example, if the optimal solution of this set of layers was an identity mapping, $f \equiv 0$. In practice, residual connections are added to bypass two consecutive convolutional layers, shown theoretically and empirically to be the best configuration [33].

## 3.4 Implementation obstacles

Even though DNN is a powerful and versatile tool, implementing models with multiple layers and parameters may be difficult in devices with limited resources. For instance, IoT networks usually have resource-constrained devices with limited memory and energy sources, in which storing multi-layered DNNs can be infeasible. An alternative for some applications is storing the DNNs remotely in a cloud server, but this can be problematic in latency-sensitive applications and does not remove energy concerns entirely, as wireless transmission can be a power-hungry application [34]. As discussed in chapter 2, many researchers are actively searching for solutions to make DNNs more accessible to resource-constrained devices. In particular, TinyML researchers seek efficient manners to implement DNNs locally, resorting to model pruning, parameter quantization, efficient coding, etc. Another alternative, and the focus of this thesis, is promoting architectural changes to the DNN so that the resulting model can satisfy resource constraints or enable fast NN inference. Constructing more compact models and resorting to EE-DNNs can assist in bringing computation closer to end devices, offering more alternatives to designers when designing DNNs for these type of problems.

An additional problem with a DNN with multiple parameters is that they are difficult to train. As discussed earlier, one of the reasons that slowed a wider adoption of NNs was shortage of training data. Moreover, if the application involves a lot of classes or instances that are hard to classify, an NN will require a significant amount of data to learn the task adequately. For instance, ImageNet [35] is a huge image dataset that, as of 2023, has almost millions of images belonging to 1000 object classes [a]. However, not every application has the luxury of having large quantities of data easily available [36], and constructing one is a laborious task that may involve labelling a large quantity of data. Hence, constructing models that are more robust to a shortage of training data is desirable.

[a] `https://www.image-net.org/download.php`

## The "costs" of overparametrization

A good way to estimate power consumption and computation latency of an NN model is by counting the number of Floating-point Operations (FLOPs) it takes for each inference [13, 37]. Let's take a fully-connected DNN layer that takes an input array with $M$ samples and produces an output with $N$ samples. Remembering equation 3.1, the linear operation is an affine transformation. A matrix multiplication between an $M \times N$ matrix and an $N \times 1$ vector takes $M \times N$ multiplications and $M \times N - 1$ additions, and if we add the bias' summation, we get a total of

$$2MN + N - M$$

FLOPs. Now, supposing that the input is two-dimensional with $P$ attributes (or channels), becoming an $M \times P$ sized matrix. Now, the matrix multiplication takes $M \times N \times P$ multiplications and $M \times P \times (N - 1)$ additions, bringing the number of FLOPs to

$$2MNP + NP - MP.$$

Hence, even though the addition of layers can enable an NN to deal with more complex data and problems, this increase can pose a significant cost to resource-constrained devices and latency-sensitive applications. Moreover, both equations show that data size plays a significant role in the number of FLOPs. Overall, these costs can be critical to the feasibility of these models' deployment, especially when we want to deploy DNNs on resource-constrained and non-cutting-edge equipment. Although convolutional layers can save computations in comparison with fully-connected layers, they can demand a lot of computations, especially when dealing with high-dimensional data. And considering that it is common to add multiple kernels in the same convolutional layer, these computational requirements can quickly stack up.

# Chapter 4

# Asymmetric Autoencoders

As seen in chapter 3, AE is a NN model commonly employed in problems that require extraction of features that characterise the core information of inserted data. In dimensionality reduction, these features summarise data and remove redundant information. In data enhancement, they enable the distinction between information we want to preserve or improve from unwanted signals and noise. Many recent applications in Internet of Things (IoT) resort to AEs, leveraging NN's data-driven approach to design models for compression and noise removal. IoT data can be challenging to traditional approaches because it can involve high volumes of data, inconsistent sample generation rates and can come from distinct sources, which can make it extremely heterogeneous [9, 23]. In particular, the development of dimensionality reduction models to minimize the amount of transmitted information is desirable because of IoT data volume and the fact that wireless transmission is a known source of significant energy consumption [9, 14, 23]. Additionally, IoT applications should be robust to noise and equipment failure, inconveniences that AE models can learn how to deal with. However, IoT network usually employs resource-constrained devices, which may not be suited for deep DNNs with multiple layers and parameters. Thus, the traditional approach of searching for performance improvements through NN depth increase will be problematic in this case. Asymmetric Autoencoder (AAE) is an alternative AE that addresses this issue by resorting to an encoder-decoder design in which the encoder has fewer layers and parameters compared to the decoder. The results compiled in this chapter show that AAEs are capable of delivering comparable performance to traditional AEs and are even capable of outperforming its symmetrical counterparts.

## 4.1 Problem description and proposal

Usually, IoT networks comprise many resource-constrained devices that resort to wireless communication to transfer their data. Wireless transmission tends to be

a power-demanding application, so solutions that minimize redundant data transmission are always in high demand [34]. Also, data can come from distinct sources, meaning that it can be extremely heterogeneous. Each device can generate data at different rates and can sense distinct phenomena. Furthermore, to keep implementation costs low, devices that compose a typical IoT can have limited capabilities, impacting the quality of collected data negatively. Because IoT data can be challenging to treat [8], many recent proposals to compress and enhance data quality that resort to AEs are being proposed [8, 9]. For example, Alsheik et al. [15] showed that a shallow AE configuration is an interesting tool that offers a lightweight compression system for resource-constrained devices, as the resulting decoder is a small set of parameters corresponding to the AE's encoder. Increasing the number of encoding layers can lead to performance improvements, much like many DNN models. In a similar problem, Ghosh and Grolinger [16] proposed an AE for dimensionality reduction to assist a Human Activity Recognition (HAR) application in which increasing the model's depth culminated in performance improvements. However, designing DNNs that will demand that a lot of layers be stored in resource-constrained devices is problematic. Considering that complex problems usually require large quantities of layers, having an alternative architecture that requires a small number of layers be stored locally can be helpful when implementing AE-based solutions in IoT networks and other scenarios that share these resource problems.

## AAE: differences and advantages of an asymmetric design

When constructing an AE for dimensionality reduction problems, the encoder will typically be stored at the node that collects and transmits data, so any increase to this structure will lead to the expenditure of more computational and energy resources. Typically, an AE is constructed symmetrically, in which the decoder configuration mirrors that of the encoder. With this approach, whenever we increase the DNN depth we are adding at least two layers, one at the encoder and another at the decoder. The AAE is an alternative to this approach in which the number of layers and other resources in the encoder is smaller than in the decoder. The key idea relies on a relevant recommendation for developing compression systems for Wireless Sensor Networks (WSNs), a subset of the IoT paradigm, that states that system design should be asymmetrical in a manner that outsources computation requirements away from resource-constrained nodes [38]. For instance, as shown in Figura 4.1, we can construct an AAE with a single encoding layer (like the one in Alsheik et al. [15]), and add as many layers to the decoder as needed to achieve our performance goals. This approach is more IoT-friendly because we can follow the heuristic of seeking performance improvements through DNN-depth increase without

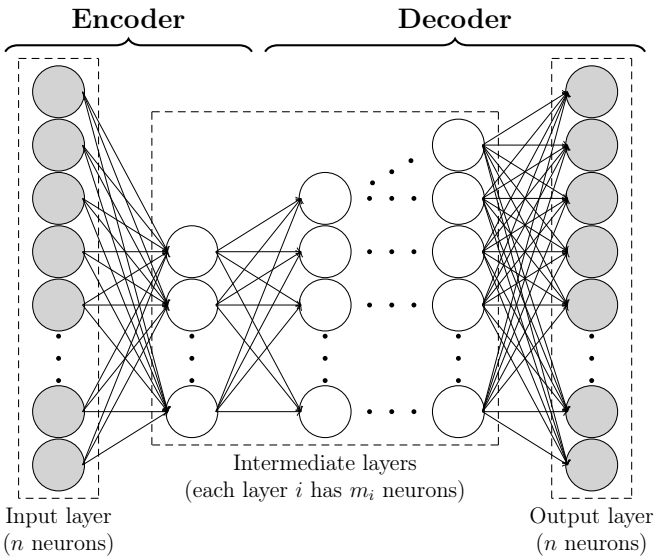



Figure 4.1: The proposed AAE. Its defining feature is having an asymmetrical design, in which the number of layers and other resources is lower at its encoder than at its decoder. For instance, we can construct an AAE with a single encoding layer, and as many decoding layers as we like.

incurring more layers to be stored in resource-constrained devices. With an AAE, we restrict ourselves to storing strictly necessary encoding layers, adding more layers to decoders to compensate for a compacter encoder. Furthermore, depth increase in an AAE occurs with the addition of a single layer. Hence, AAEs have fewer layers and parameters than a symmetric AE with similar decoding depth, which can make them easier to train than its symmetric counterpart.

The proposed AAE is deployed much like most AE-based solutions in IoT, where encoding layers are placed in the resource-constrained nodes and decoding layers are stored at a remote server [9, 15], the destination of the data transfer operation. For example, in a sensing application, nodes in a typical WSN benefit from this setup to maintain their operation costs low. The model is trained offline using historical data of the phenomenon we want to monitor. To enable the outsourcing of computation away from the resource-constrained devices, it is assumed that the destination has enough resources to store and perform the computations required by the decoder.

## 4.2 AAEs for stream-like data compression

In the following experiments, AEs are designed for stream-like data compression, in which a number of samples are collected in a single sensing node for a period of time before being transmitted to its destination. The DNNs are trained to learn how to extract intrinsically temporal correlations from the collected data to generate a compressed representation. The experiments are divided into two main sets, one

involving noiseless compression and another that requires data compression with noise removal. Altogether, the experiments show how increasing the model's depth impacts reconstruction error, and how the limited amount of training data impacts the deeper models' training. As discussed in chapter 3, lack of training samples can be a common issue when training DNNs to deal with time-series data [36]. This can be problematic when training models to handle complex tasks, which usually require more training data. The experiments with noisy data enable this investigation, as the problem becomes more complex than the first set of experiments because now models need to learn how to perform data compression together with noise removal. AAEs are fully-connected or can contain convolutional layers (Convolutional AAEs (CAAEs)). Specifically, the latter group of AAEs shows the advantages of adding NNs suited for exploiting data particularities, which in the following experiments are short- and medium-term relations. Overall, the results show that AAE are capable of delivering comparable performance to AEs. Moreover, it shows that the proposed DNN is even capable of outperforming the symmetric AE, and is more robust to shortage of training samples.

### 4.2.1 Experiment setup

The models are evaluated using temperature readings from a wireless sensor node from the American River Hydrologic Observatory (ARHO). Specifically, a node located near Caples Lake, in California. Samples were taken uninterruptedly from June 2014 to October 2017, in regular intervals of 15 minutes. Usually, IoT datasets are often challenging to handle due to inconsistent data generation rates and variable data quality, which may hinder the interpretability of initial experiments. Preliminary experiments identified the value of using the selected dataset, and later introducing data irregularities into it, to aid result interpretation. All AEs (AAEs included) are designed to take an array of 100 samples as input, and compress them down to an array of 25 values. Thus, the sensor node forwards data every 1500 minutes or roughly a day's worth of samples. This is not a result of the AEs computation, but rather a limitation of the dataset adopted. Adopting a shorter period between consecutive samples is sufficient if one desires a faster transmission rate. Specifically, in the data compression with noise removal experiments, an array of samples can be corrupted with Additive White Gaussian Noise (AWGN). Noise levels can vary from 2.5 to 80 db Signal-to-noise ratio (SNR), and are randomly drawn in the training phase. As the experiment results will demonstrate, training AEs to perform data compression in the presence of noise compared to noiseless conditions. This difficulty arises because the models must now learn both tasks concurrently, highlighting the relation between DNN size and training sample availability.

All models are trained using the same methodology. First, samples up to June 2016 were split to form the training set, whilst the remaining samples formed the testing set. This means that approximately 60% of samples are used for training, and the remaining 40% for testing. Additionally, training arrays were constructed using a sliding window method to increase their numbers, adopting strides equal to 10, 17 and 23. These values were selected to maximize the number of unique training arrays, as these numbers are coprime. After this process, repeated arrays were discarded. These decisions were adopted after early experiments indicated that training with such a set allowed the models to generalize well. In addition to this, models receive batches of 50 arrays at each training iteration, and their reconstruction errors are measured using Mean Squared Error (MSE).

The Nesterov-acelerated Adam (NAdam) optimiser is used together with a reduce Learning Rate (LR) on plateau scheduling [a] and early stopping. NAdam is a variation of Adaptive Momentum Estimation (Adam) optimiser [39] that adopts Nesterov momentum [40] to speed the Stochastic Gradient Descent (SGD) convergence [41]. Reduce LR on plateau is a scheduling policy that decreases the LR when a model stops improving after a period of epochs [b] since the last best parameter configuration was found. Using the validation set, each model performance is measured after each epoch and the configuration that produced the lowest reconstruction error is saved. In the experiments that follow, this period is set to be equal to 5 epochs, after which the LR is reduced to 75% its original value, and the starting LR was defined empirically for each AE configuration. Finally, the early stopping policy consists of finishing the training process after the model stops improving. The early stopping waiting period is set to 15, and its evaluation starts after the last update of the reduce LR on plateau (when the smallest LR is achieved).

The code used in the experiments is written in Python, built using the `TensorFlow` framework [42]. It is made available at a `GitHub` repository [43]. Each model is trained 10 times, using different initial weights. The performance of each model is evaluated by computing the average reconstruction error over all testing arrays, and the reconstruction error of each array is measured using the MSE. Finally, in the case of the models trained to perform noise suppression, we assess their robustness to noise using AWGN and the same SNR levels.

### 4.2.2 AAE configurations

All models are designed to reduce 100 temperature readings to a compressed array of size 25. AAEs encoders consist of a single layer that is responsible for the

[a] `https://keras.io/api/callbacks/reduce_lr_on_plateau/`

[b] An epoch refers to a single pass through the entire training set, encompassing all the available samples.

aforementioned compression. All hidden layers adopt Scaled Exponential Linear Unit (SeLU) as the activation function, whilst the output layer activation function is the sigmoid function, the latter a common choice for AEs. Because of this, each sample array is normalized to fall within the sigmoid codomain. Consequently, the maximum and minimum values of the corresponding batch are also transmitted with the compressed data for reconstruction.

All fully-connected decoder configurations exhibit a gradual increase in the number of neurons per layer as they approach the output layer. As seen in Table 4.1, arrows are used to highlight the data flow direction for all decoder configurations. Fully-connected AEs are labelled using the notation "Autoencoder-Model-DepthOfTheVaryingBlock", adopting the decoder's depth as the index of each model. For example, the first AAE architecture (AAE-1) contains a single hidden decoding layer between the decoder's input (layer with 25 neurons) and output (100 neurons). The symmetric AEs, against which we compare the proposed AAEs, have encoders that mirror the configurations seen in Table 4.1, as in Figure 3.2. In the case of AE-0, it refers to the shallowest symmetrical AE, which has only two NN layers, one for encoding and another for reconstructing data. This is the base model, being the simplest and starting model for evaluation, which is modified to achieve performance enhancements.

Table 4.1: AEs and AAEs fully-connected decoder configurations

| **Label** | **Layer sizes** |
|---|---|
| AE-0 | $25 \rightarrow 100$ |
| (A)AE-1 | $25 \rightarrow 50 \rightarrow 100$ |
| (A)AE-2 | $25 \rightarrow 50 \rightarrow 75 \rightarrow 100$ |
| (A)AE-3 | $25 \rightarrow 45 \rightarrow 65 \rightarrow 85 \rightarrow 100$ |
| (A)AE-4 | $25 \rightarrow 40 \rightarrow 55 \rightarrow 70 \rightarrow 85 \rightarrow 100$ |

### 4.2.3 CAAE configurations

Intrinsic temporal correlations are an attribute present the data used in the previous experiments, and are underused in the fully-connected models. Although fully-connected AAEs exploit them to to a certain degree, their usage is enhanced thanks to the inclusion of convolutional layers. These layers are added to the decoder to encourage the utilization of short- and medium-term relations in the reconstruction process to a greater extent than in the previous AAEs. This approach aims to leverage prior knowledge of the sensed signal to achieve performance improvements. To distinguish it from the previous AAE, and because it has convolutional layers, it is referred to as CAAE.

All CAAEs proposed analysed in the following experiments are constructed modifying AAE-2 (seen in Table 4.1), replacing the layers that perform $50 \rightarrow 75$ and $75 \rightarrow 100$ transformations with convolutional layers. Another feature of the CAAE is that no convolutional layer performs upscaling operations. Instead, data is upsampled by a factor of 2 before the last convolutional block (*i.e.*, set of convolutional layers). Finally, all data from the last convolutional block are fused together using a convolutional layer with kernels with both size and stride equal to 1. This layer is the CNN's output, thus it also uses sigmoid as its activation function.

Table 4.2 brings the convolutional block configurations of each CAAE. Similar to Table 4.1, the first index in the identifier reflects the decoder's depth. Differently, the second index reflects the complexity of the convolutional layers, with the higher identifier reflecting the models with more kernels. Additionally, all kernels have a size equal to 3, and the convolutional blocks can have an expansion factor of 4 or 8. This value represents the amount of spatial convolutional filters applied at each input channel. Hence, for $n$ input channels, if the block has $t$ expansion factor, the total number of kernels is $nt$. If $nt$ is different from the number of output channels, inter-channel mixing is applied. Similarly to the hidden layers of the previous AAEs, we employ SeLU as the activation function of intermediate layers. Finally, regarding CAAE-4.$x$ models, the second convolutional block is an HDC.

Table 4.2: CAAE configurations

| | $1^{st}$ **Conv. Block** | | | $2^{nd}$ **Conv. Block** | | | $3^{rd}$ **Conv. Block** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Label** | **nout** | **exp** | **dil** | **nout** | **exp** | **dil** | **nout** | **exp** | **dil** |
| CAAE-3.1 | 4 | 4 | 1 | 4 | 4 | 1 | - | - | - |
| CAAE-3.2 | 8 | 4 | 1 | 8 | 8 | 1 | - | - | - |
| CAAE-4.1 | 4 | 4 | 1 | [4, 2, 2] | [4, 4, 4] | [1,2,3] | 8 | 4 | 1 |
| CAAE-4.2 | 8 | 4 | 1 | [4, 2, 2] | [8, 8, 8] | [1,2,3] | 8 | 8 | 1 |

## 4.3 A resource-constrained friendly way to increase AE's depth

When working with DNNs, a rule of thumb when seeking performance improvements is increasing the number of NN layers. However, this approach may be limited or not possible when working with constrained devices. In the experiments that follow, we will see this trend of improvement with deep symmetric AEs, and how the proposed AAEs can accompany their symmetric counterparts. Moreover, the results show that they can outperform the symmetric models while requiring fewer FLOPs.

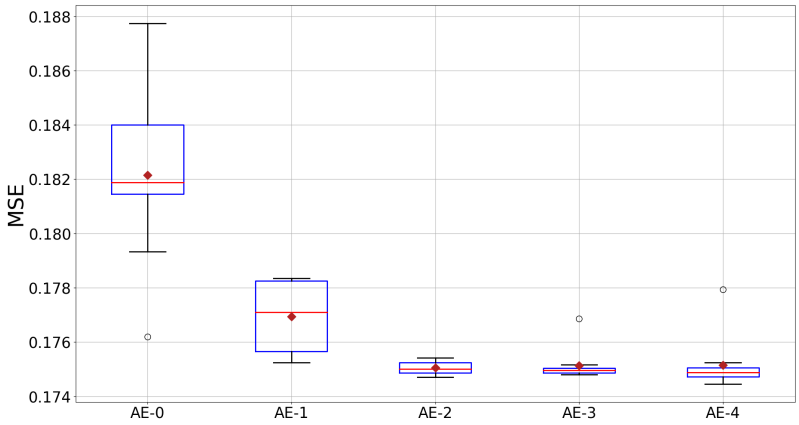



Figure 4.2: Mean reconstruction error for all symmetric AEs.

### 4.3.1 Increasing AE depth

Figure 4.2 brings a boxplot showing the distribution and the mean reconstruction error for all analysed symmetric AEs. It clearly shows the advantages of increasing the number of layers, especially compare AE-0 with the deeper configuration. Apart from a single model that delivered a mean reconstruction error close to 0.176 MSE, all AE-0 models were unable to outperform any of the deeper AEs. Moreover, the models that delivered the lowest reconstruction error are those with at least seven NN layers (AE-2 and deeper configurations). However, when observing the median (red horizontal line segment) and the mean (dark red diamonds) of these models, it is noticeable that there is no significant gain when opting for a model with more than two hidden decoding (and encoding) layers. Notice that, by increasing the number of layers symmetrically, we are adding two more layers compared to the previous configuration, adding one layer at the encoder and another at the decoder. Hence, the deeper models have a significant amount of parameters to adjust, requiring more training data. This is another problem that may occur when deploying deep symmetrical-AE architectures in IoT networks, as the amount of data on a particular phenomenon can be limited. As will be seen later with our proposed models, this issue is minimized significantly.

Table 4.3: Number of parameters and FLOPs needed for each compression in each AE encoder.

| **Conf.** | Numb. Params. | FLOPs |
|---|---|---|
| AE-0 | 2525 | 5025 |
| AE-1 | 6325 | 12575 |
| AE-2 | 12650 | 25150 |
| AE-3 | 18295 | 36370 |
| AE-4 | 21775 | 43275 |

Before moving on to the AAEs, it is important to quantify the increase in size

of the encoders. Table 4.3 brings the number of trainable parameters (weights and biases) in the encoder of each AE configuration. The number of parameters grows considerably as the number of layers increases. For instance, to achieve the performance gains in Figure 4.2, the number of encoding parameters more than doubles from AE-0 to AE-1, and doubles going from AE-1 to AE-2. Moreover, this issue tends to worsen when dealing with more complex data, which requires more layers and neurons. The proposed AAEs are an alternative to minimize these problems. Even if quantization is adopted later, starting with a model that is already more compact can lead to even more space conservation. For instance, this can be beneficial when envisioning that multiple models can be stored in a sensing node to deal with data collected from multiple phenomena [23]. Current quantization methods can reduce a maximum of 4 times the original size at best, meaning that we can still face storage and energy problems when adding multiple DNNs in a single resource-constrained node.

Another issue with increasing AE size is the increase in encoding computations. Table 4.3 brings an approximation of FLOPs for each encoder (ignoring the FLOPs that SeLU may impose). As discussed in chapter 3, this can give us a good estimate of the number of computations needed to generate one compressed array. For instance, when opting for AE-1, the amount of FLOPs more than doubles in comparison with AE-0. Moreover, if we opt for AE-2, which was one of the best-performing models, the resulting encoder imposes FLOPs that are more than five times higher than those of AE-0. Adopting a similar analysis as Alsheikh et al. [15], which computed the power consumption in an MSP430 microcontroller, these increases in FLOPs lead to about 4.02 $mJ$ and 10.71 $mJ$ more energy spent in each data compression (appendix 4.A). AE-0 spends 2.65 $mJ$ per compression. Thus, opting for AE-1 nearly triples power consumption, and AE-2 spends more than five times more energy. This trend tends to worsen with more complex data because they will likely require models with more neurons and encoding layers. Additionally, other factors, such as sampling rate, can increase sensor node power consumption. The proposed AAE addresses this computational problem by showing that performance improvements are possible even when we keep AE-0's encoder configuration, showing that these implementation issues can be mitigated through a different AE design.

### 4.3.2 AEs vs. AAEs

Figure 4.3 shows that the AAEs keep the trend of improvement as we increase the number of layers. Recall that the AAE have the same encoder configuration as AE-0, meaning that they have the same number of encoding parameters and

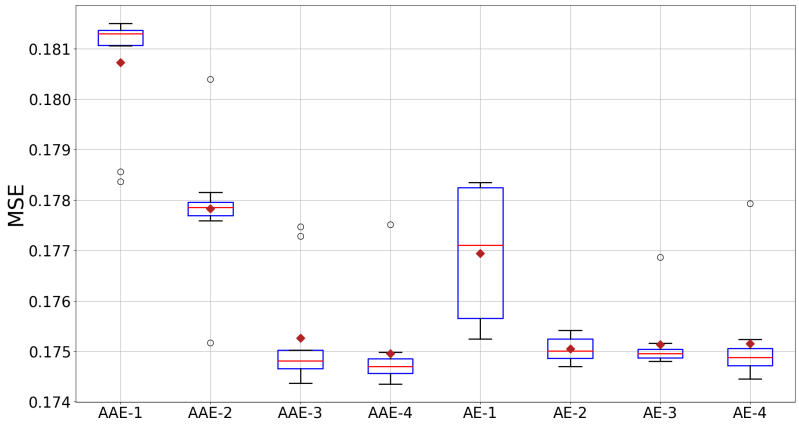



Figure 4.3: Comparison between the mean reconstruction error distribution of the proposed AAEs against their symmetrical counterparts.

impose the same number of FLOPs shown in Table 4.3. The additional layers are inserted in the decoder, to be implemented away from the resource-constrained devices. However, it is noticeable that the advantage of the asymmetric design starts to appear with AAE-2, model with three decoding layers. Interestingly, this is the AAE with the same number of NN layers as AE-1 (both have 4 NN layers in total), and they have comparable performance. This suggests that, at least for this and similar scenarios, the number of NN plays a more significant role than encoder-decoder layer disposition. For example, it is noticeable that their medians and means are less than 0.001 MSE apart. Considering computation demands, AAE-2 is more advantageous for resource-constrained implementations because it requires 5025 FLOPs per data compression, whilst AE-1 takes 12575. This is less than half the computation needed in the symmetric model.

Moving on to the deeper architectures, the advantages of the proposed AAEs become more evident. AAE-3 and AAE-4 not only have comparable performance to AE-2 and deeper configurations but are also capable of outperforming them while requiring five times fewer parameters and FLOPs, at least. First, they are capable of delivering a similar performance, evidenced by the observation that AAE-4 and AE-4 have similar error distributions, with AAE-4 having a slightly better performance. For instance, the third quartile of AAE-4 (topmost side of the box) is almost equal to the AE-4 median. Additionally, we can observe that some deeper AAEs are capable of outperforming most of all trained AEs. Observe in the graph that the median of both AAE-3 and AAE-4 are below the medians of AE-3 and AE-4. Given that the AAEs have an encoder with the same size as AE-0, they appear a better choice for the analysed scenario. Opting for either of the two AAE configurations instead of AE-3 and AE-4 means that we are saving seven times fewer FLOPs per compression operation, at least. Thus, AAEs can be a valuable alternative for resource-constrained devices.

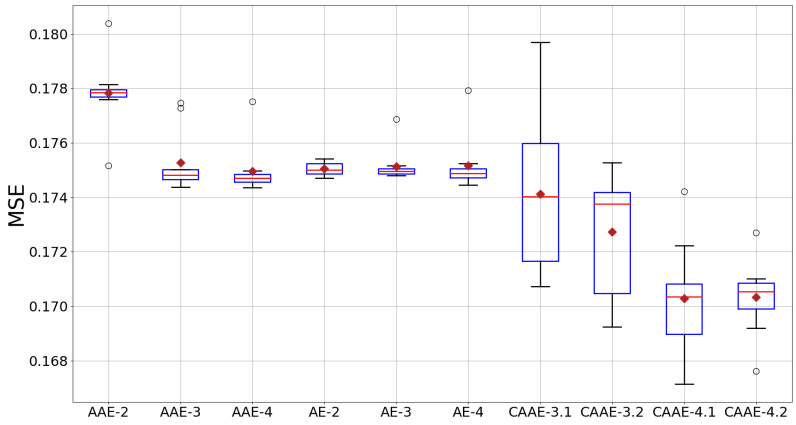



Figure 4.4: The mean reconstruction error distribution of the proposed CAAEs compared with the best-performing AEs and AAEs.

### 4.3.3 Exploiting temporal correlation with CAAEs

Figure 4.4 shows the performance of the CAAEs, comparing them with the best models so far. These results show the advantages of adding convolutional layers to the asymmetrical architecture. To illustrate this, notice that the models achieving the best results among all configurations are constructed with HDC blocks, and that even the worst-performing CAAE models outperform all AEs and AAEs seen previously. This is another positive result for the proposed asymmetrical architecture, emphasizing that we can seek performance improvements by adding new features to the decoder that absent in the encoder. However, Figure 4.4 shows that the results obtained with CAAEs have more variance than those obtained with our previous architectures. For the sake of fair comparison, we adopted the same training parameters for all models. Hence, it is possible this variance can be attributed to poorly defined training parameters, and it is possible to reduce this variance by adopting more appropriate training parameters.

Analysing solely the CAAEs results, the fact that the models with an HDC block present the best results highlights the advantages of employing a layer with distinct dilations. By doing so, we ensure that the kernels learn to extract distinct features because the HDC kernels are operationally distinct. Moreover, these features are complementary, given they are obtained in different resolutions. We know that temporal data have short- and medium-term relations, meaning that a CAAE with HDC is more suited to extract them than a CAAE that don't have this convolutional block.

When comparing CAAE-4.1 and CAAE-4.2, we observe that adding more convolutional filters (through the expansion factor) seems unnecessary. This appears similar to the results in Figure 4.3, when comparing the deeper AEs and AAEs configurations. This suggests a ceiling in performance or indicates the occurrence of an

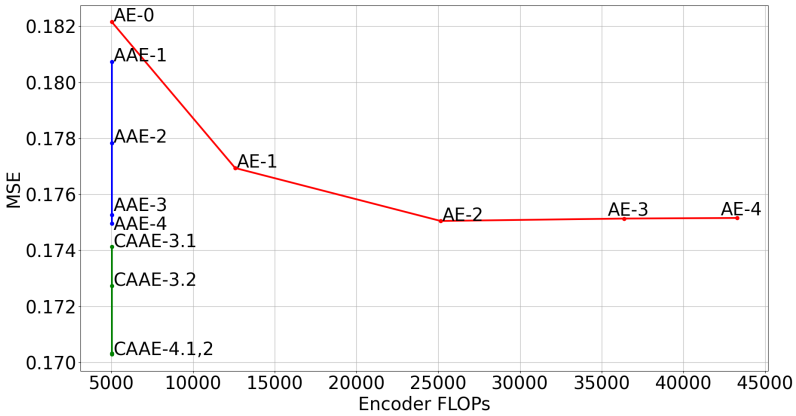



Figure 4.5: Comparison between proposed AAEs and AEs reconstruction errors and encoder FLOPs demands. Performance improvements are possible without increasing the encoder's FLOPs, with CAAEs outperforming the deep AEs.

overfitting due to lack of training data. However, as will be seen in our next experiments with noise, this can be attributed to the problem's relative simplicity rather than a shortage of training data. For example, increasing the number of kernels is advantageous when comparing the performances of CAAE-3.1 and CAAE-3.2, which have convolutional kernels with the same dilation. Compression without denoising is a more straightforward task, suggesting that having a lot of convolutional kernels becomes unnecessary when deploying the HDC block, given its improved capacity to extract distinct and complementary features.

### 4.3.4 Reconstruction error and encoder requirements

In addition to results shown previously, Figure 4.5 shows the mean reconstruction error of each AE configuration, considering the number of FLOPs needed for each data compression. In it, we can see the advantages of opting for the asymmetrical approach. By bringing the encoding computation values compiled in Table 4.3, we can put the reconstruction results in perspective to the encoding demands of each AE. First, we observe that adding more layers and features to the AAE's decoders enhances the model's performance without incurring more FLOPs. Additionally, as the CAAEs results show, we can add new features to the decoder to improve data reconstruction, allowing us to outperform the symmetric and costly AE. Overall, combining these results with the ones previously discussed, we can see that AAEs can offer designers a more compact model capable of rivalling the deep AE configurations without increasing encoder implementation costs, which can help its implementations in resource-constrained devices.

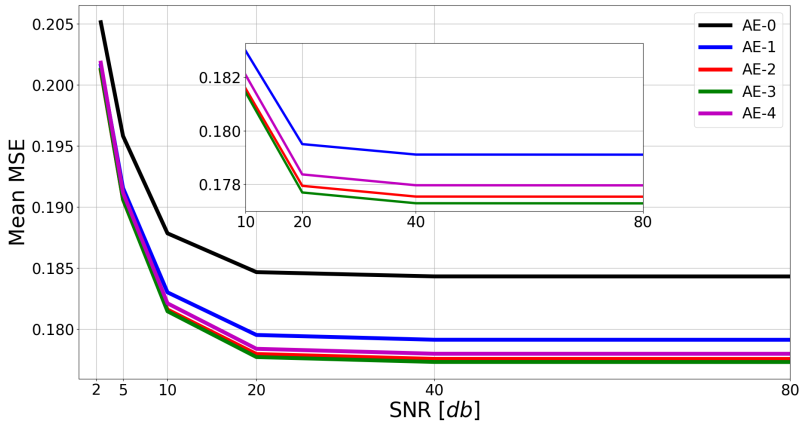



Figure 4.6: Mean reconstruction error for all symmetric AE.

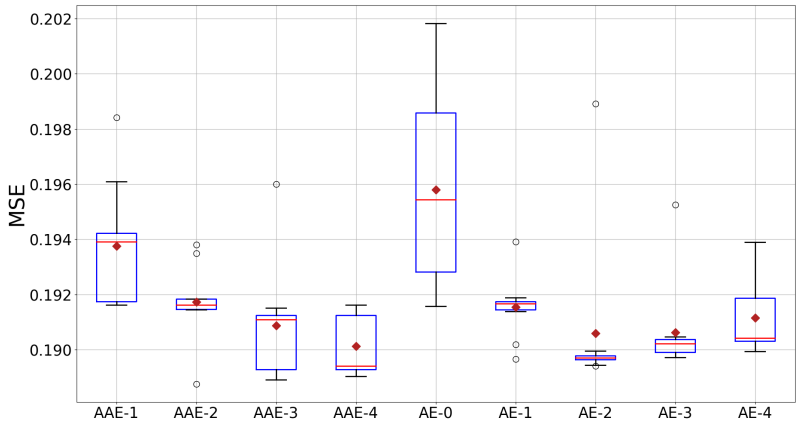
## 4.4 Dealing with noisy data

Requiring that AEs are also capable of removing noise from data can impose a challenging scenario for the models to be trained. Not only must all AEs learn how to perform data compression, but also they must learn how to perform noise suppression. Our experiments show that this is particularly challenging for NNs with multiple trainable parameters, which need a lot of data to adjust their parameters. Hence, in this scenario, data shortage problems, which was an issue when training models with multiple layers, tend to be aggravated.

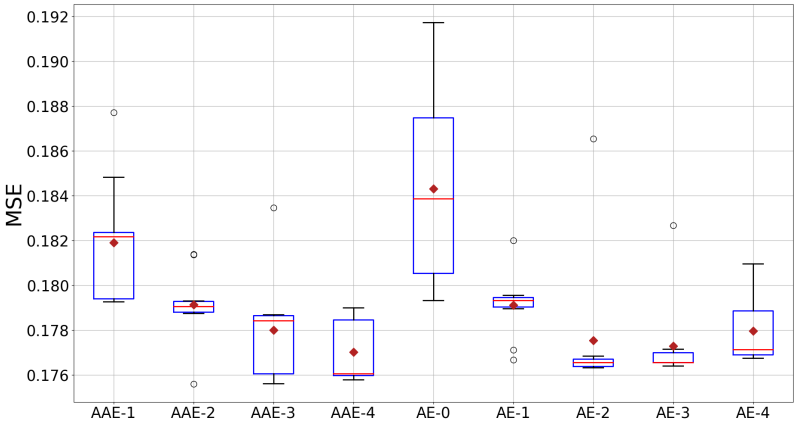
### 4.4.1 Increasing AE depth

Figure 4.6 shows the mean reconstruction error for all symmetric AEs. The first noticeable result in this figure is AE-0 being outperformed by all other AEs, and by a considerable margin. For example, even when AE-0 is subject to high SNR levels, it offers a higher reconstruction error than the AEs deeper than it in worse noise conditions. All AEs except AE-0 deliver a mean error below 0.185 MSE under SNR levels equal to 10 db, whereas the base model's mean error at 80 db is almost equal to this value. Later, when comparing these symmetric AEs with AAEs, it becomes clear that the proposed models are fundamental to enabling AEs with a single encoding layer to match or even surpass the performance of models with multiple encoding layers.

The second noticeable result is that AE-2 and AE-3 outperform AE-4, which is the deepest AE configuration. The difference is even more significant when comparing AE-3 with AE-4. Figure 4.10 brings a more in-depth analysis of the AE performance under SNRs equal to 5 db and 40 db, showing that most of the models constructed with the AE-2 architecture deliver better reconstruction errors than the deeper symmetrical models. This is particularly pronounced in the scenario with




(a) Reconstruction error with SNR=5 db.




(b) Reconstruction error with SNR=40 db.

Figure 4.7: Reconstruction error for all fully-connected autoencoders.

more noise, as most AE-2 models are among the best-performing symmetric models (Figure 4.7(a)). Notice that under 5 db SNR, nearly all AE-2 models deliver a lower reconstruction error than all AE-4 models, and surpass at least half of the AE-3 models, given that they are below AE-3's median. We can attribute this as being another limitation of employing deep NN architectures to deal with problems that have low availability of training data. The scenario at hand here is more challenging than the previous noiseless experiments because now models must also learn how to remove noise from the sensed signal. This suggests that more samples are needed to train deeper models to perform both tasks simultaneously. Because the proposed AAE usually have fewer adjustable parameters than the symmetric models, the necessity of more training samples is another issue that can be worked around.

### 4.4.2 AEs vs. AAEs

Figure 4.8 shows the mean reconstruction error for the proposed AAEs (continuous lines) in comparison with their symmetric counterparts (dashed lines). Focusing solely on the models with 2 hidden decoding layers or less, we can observe that shal-

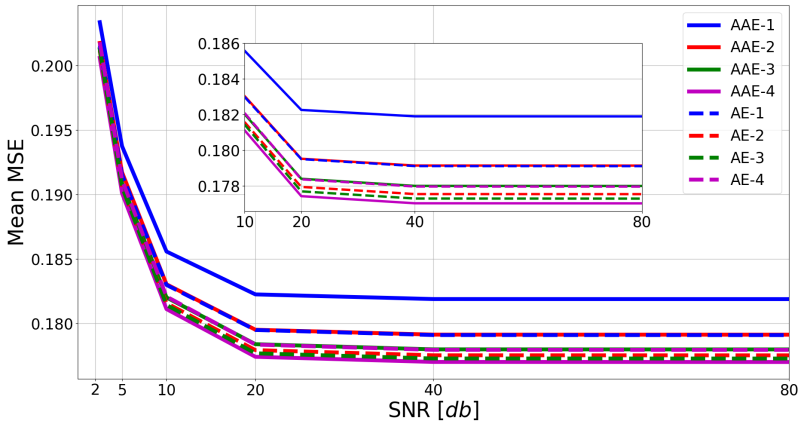



Figure 4.8: AAEs mean reconstruction error results compared with the symmetric counterparts. Models with the same colour scheme have the same decoder layout.

low AAEs do not fall behind their symmetric counterparts. Furthermore, AAE-2 was able to deliver a performance nearly identical to AE-1. Remember that these models have the same number of layers, which makes the former a more sensible choice for a resource-constrained implementation. Additionally, unlike observed with AEs, the deeper models keep the improvement trend seen in the noiseless scenario as the number of layers increases. The error obtained with a deep AAE is always below shallower configurations across all SNR levels above 5 db, which was not the case with the AEs. This is more evident when we observe the boxplots in Figure 4.7. Moreover, noticing that the median of AAE-4 in both plots is below all configurations, we observe that the deepest AAE architecture outperforms all AE models. These results suggest that the proposed AAEs is more robust to training sample shortage discussed earlier, and that they also fit well in IoT compression scenarios that need noise suppression. Another result shown in Figure 4.8 is that for SNR ranging from 20 db to 80 db, the performance of all models is seemingly the same, that is, there is no significant performance degradation as we increase the noise levels. Also, the performance drop from 20 db to 10 db is not very eminent, in comparison with the drop from 10 db to 5 db and from the latter to 2.5 db. Both results suggests that the proposed training methodology is successful. However, we expect that the adoption of regularizers to assist the AAEs in learning denoising can improve results for higher noise levels.

Figure 4.7 shows that the relative performance of all models has minimal changes when the noise level increases. The overall performance of all AEs and AAEs is similar in both graphs, and there is also consistency in the variance among AEs that have the same architecture. This suggests that models that perform well at low noise levels tend to deliver the smallest reconstruction error in worse noise conditions. However, as discussed earlier when AE-2 performance at 5 db was compared with the deeper AE models, the loss in performance is more pronounced with the deeper

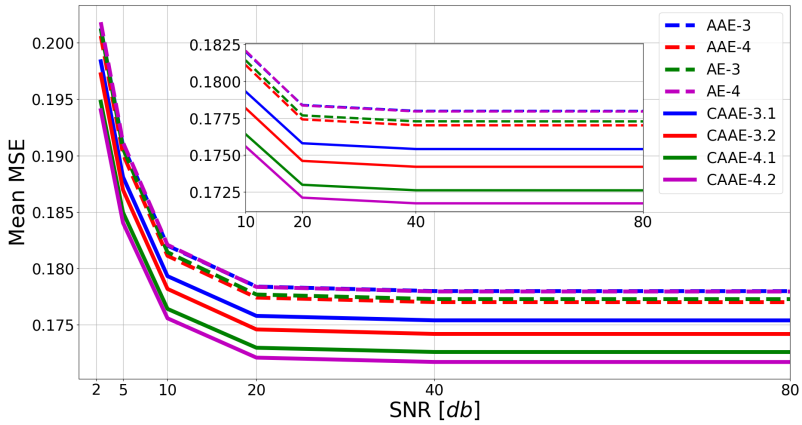


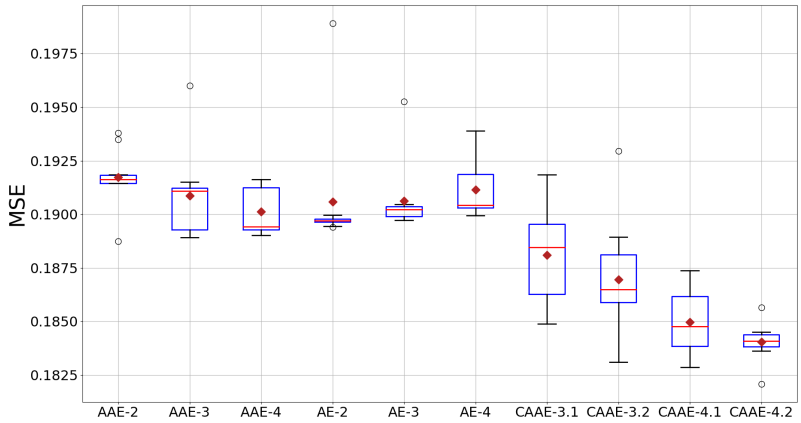

Figure 4.9: CAAEs mean results, in comparison with the AAEs and AEs with similar decoder depth.

AE architectures. This is not an issue with the AAEs, as the deeper models that performed well at high SNR levels still outperform the other AAEs when the SNR levels are lower. This suggests that deeper AAEs are less sensitive to noise than their symmetrical counterparts. Thus, one can expect that AAEs are well-suited to concurrently perform the compression and denoising tasks simultaneously. Besides having an encoder with fewer parameters to store and computations to perform, the fact that they have fewer layers overall in comparison with the symmetric ones means that they require fewer training samples to adequately learn complex tasks.

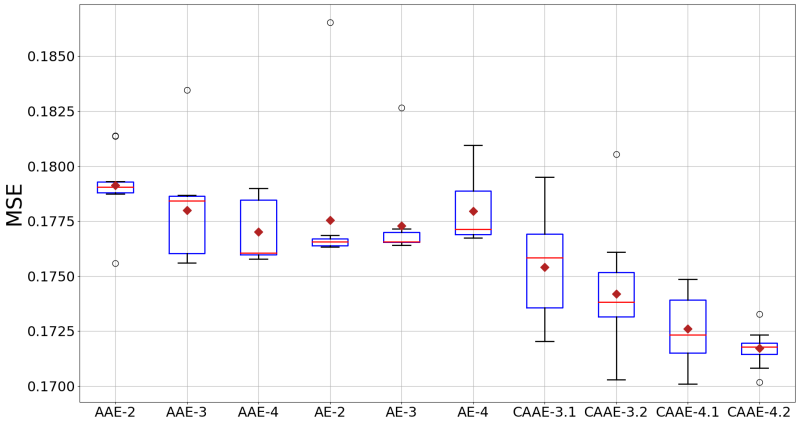
### 4.4.3 Exploiting temporal correlation with CAAEs

Figure 4.9 shows the mean reconstruction error of the proposed CAAEs in comparison with the AEs and AAEs with similar decoder depth, *i.e.*, models with three or more hidden decoding layers. Note that the CAAEs keep the performance improvement trend seen with the other AAEs. Furthermore, combining these results with those found with the other asymmetric models, and noticing that they all have the same encoder configuration as AE-0, it is noticeable that the main contribution of the proposed asymmetric architecture is opening new opportunities for performance improvement that don't rely on more encoding layers. So, by constructing decoders with enough resources and capable of exploiting the distinctive features of the phenomenon of interest, one can design AEs with a limited number of encoding layers that can overcome the lack of encoding layers.

Figure 4.10 presents a detailed analysis of the CAAEs alongside the best-performing AE seen previously. Similarly to Figure 4.7, the overall dispersion of the CAAEs under different noise levels is almost unchanged. This is another result that backs our assumptions that asymmetric autoencoders performing well with low noise levels tend to be the best models when the level of noise increases. How-




(a) Reconstruction error with SNR=5 db.




(b) Reconstruction error with SNR=40 db.

Figure 4.10: In-depth look of best-performing autoencoders.

ever, Figure 4.10 brings an additional result when compared with Figure 4.4, which showed CAAEs performance with noiseless data. Different from the latter, where adding more kernels seemed irrelevant, CAAE-4.2 benefits from the greater number of filters. This suggests that one can experiment with adding more kernels to the convolutional blocks, to assess if more filters can improve the CAAEs performance in a more complex task. Given that these changes are done to the decoder, the resource-constrained device where the encoder will be implemented is oblivious to changes in the CAAEs design.

## 4.5 Conclusion

An important concern when implementing DL-based solutions in IoT networks is the burden they may pose to sensing and actuator nodes. Typically, NN models require significant storing and computational resources that resource-constrained devices may not provide. Therefore, there is a need to explore alternative NN architectures. Recently, many solutions that resort to AE have been proposed to

handle IoT data. Among them, AEs were shown to be useful in data compression and noise suppression applications. However, the typical AE architecture does not scale well for IoT deployment, as increasing the number of encoding layers poses a problem for implementing AE-based solutions in resource-constrained nodes. Hence, to alleviate NN's implementation in this and similar scenarios, AAE is proposed as an AE variation in which the number of layers (and other computational resources) is greater in the model's decoder than in the encoder. This offers an alternative that can shift the bulk of the computation away from IoT nodes.

The results show that the adoption of the proposed AAEs suits well IoT sensing. These asymmetric models have encoders with fewer parameters and that require fewer FLOPs per data compression operation. The AAEs are capable of delivering reconstruction errors that can rival their symmetric counterparts, and can even outperform them. Additionally, AAEs seem to be more suited for the scarcity of training samples, a problem that may appear in many IoT settings. This assertion is based on results that keep the performance improvement with depth increase in both scenarios analysed, whereas symmetric AEs fail to keep this improvement trend. Especially when trained to perform denoising together with data compression, the deeper AE models stopped showing significant improvements, likely due to the shortage of training samples. Furthermore, the results of the CAAEs show that we can pursue further performance improvements by modifying the decoders to exploit the particularities of the phenomenon of interest. Given that the signal used in the experiments has temporal correlations, the reconstruction error of the recovered signal was minimized by adding convolutional layers to the decoder. Hence, the main result presented in this chapter is that one can construct encoders with a limited number of layers and improve feature extraction by adding more resources to the decoding block and exploiting prior knowledge about the sensed signal. This offers an alternative approach that can be helpful when designing AE-based solutions for resource-constrained devices in IoT networks and other scenarios that share similar problems.

## 4.A Energy Consumption Estimation

According to Alsheikh et al. [15], one clock cycle in an MSP430 microcontroller accounts for 1.85 $nJ$. Additionally, each multiplication and addition operation requires 395 and 184 clock cycles, respectively. A matrix multiplication between an $M \times N$ matrix and an $N \times 1$ vector takes $M \times N$ multiplications and $M \times N - 1$ additions. Hence, in each encoding layer, we have

$$(395MN + 184M(N-1) + 184N) \cdot 1.85\ nJ.$$

Thus, recalling the encoder configurations in Table 4.1,

- AE-0 consumes approximately 2.65 $mJ$;
- AE-1 consumes approximately 6.67 $mJ$;
- AE-2 consumes approximately 13.36 $mJ$.

# Chapter 5

# Semantic Segmentation with Early-Exit DNNs

Usually, a deep DNN is needed to address complex problems. An NN with more layers and more powerful computing features can extract and construct complex features. Thanks to mechanisms like residual connections, constructing models with hundreds of layers is a reality [33], and we can expect that constructing DNNs with many more layers will be possible in the future thanks to the continuing advances in NN theory. Advances that can lead to new layers and model configurations, which themselves can lead to deeper and more complex implementations. However, deep DNNs can be difficult to implement in resource-constrained and latency-sensitive scenarios [6, 13, 20]. An alternative to address these issues is to lower implementation complexity through model compression, quantization, and pruning (see TinyML in chapter 2) at the expense of reduced inference precision [20]. Another alternative lies in the fact that simpler data can be classified using features extracted in layers close to the DNN's input, making further processing in the remaining layers unnecessary [12, 20]. Thus, constructing a model that takes this into account can be beneficial.

Early-Exit DNN (EE-DNN) is a multi-output NN implementation that allows simpler input data to be inferred using features extracted in early layers. Together with DNN partitioning, these models were shown as a valuable tool to develop image classification in edge-cloud co-inference [13, 22], where applications resort to cloud resources only when strictly necessary. Thus, it is only natural to try to replicate this success in other CV problems. However, inserting early exits in CNNs, backbone of almost all CV-related DNNs, can be challenging because early layers work with high-dimensional data [20]. In the case of semantic segmentation, these issues can be aggravated because DNNs need to extract features in multiple resolutions and preserve localized information. The following experiments show how to construct and train EE-DNNs to perform semantic segmentation, and how useful these exits can be to applications like autonomous driving. Moreover, the qualitative results show

the usefulness of EE-DNNs in semantic segmentation, as even coarse segmentations can be good enough to supply time-sensitive applications.

## 5.1 Problem description and proposal

CV is an area that involves designing models for image-like data processing to perform tasks involving picture captioning, object classification and tracking, image segmentation, etc. ML models, and in particular DL, are very successful in CV problems because they can automate the process of selecting feature extractors [25], making the design CV applications more accessible to non-experts in image processing. Additionally, DNNs are now state-of-the-art in many CV datasets and applications [1, 25]. Because of these reasons, many areas like autonomous driving, smart health and environment monitoring resort to DL-based solutions to design their CV-based applications [1]. Moreover, thanks to the recent abundance of cameras in urban environments, paradigms like vision-aided wireless communication [7, 44] aims to incorporate CV-generated data to help urban and infrastructure applications (in the example, communication applications). Incorporating visual data in applications where they are now absent or underutilized can enhance information quality and diversity, leading to better decision-making.

Among CV problems, semantic segmentation is concerned with pixel-level classification with which we can segment an image into identifiable instances, identifying which elements are present in an image and determining their respective areas [1]. When compared with image classification, it moves from identifying if a certain class is present to partitioning an image into regions, each belonging to a class. For example, Figure 5.1 shows an example of a semantic segmentation application that has to identify objects of interest within an image, identifying their position and the area they occupy. If given the picture seen in Figure 5.1(a), a model trained in semantic segmentation to identify people and vehicles in the foreground needs to distinguish between cyclists and their bicycles. Moreover, each pixel should be assigned to a class of interest so that collectively identify the objects found. Hence, if designed correctly, the model produces the image seen in Figure 5.1(b).

Because we shift from traditional image classification to pixel-level classification, training a DNNs for semantic segmentation can be more challenging than the former for at least two reasons. First, it requires that feature maps carry localized information [31]. This means that each layer must learn how to extract and preserve information that links each feature to the location it belongs in the image. In DNNs this is particularly important for information extracted in early layers, as they can be lost as data moves towards the DNN's output. Hence, besides learning which features are relevant and how to extract them at each NN stage, models must

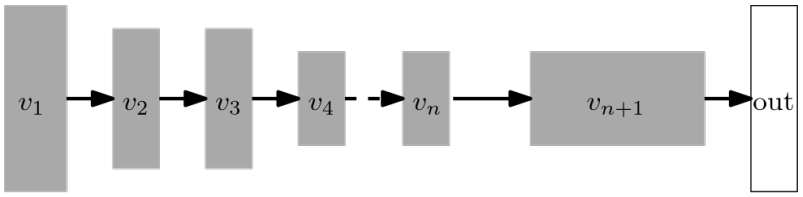
(a) Original image. (b) Segmented image.

Figure 5.1: Example taken from Pascal VOC 2012 dataset [45] showing how semantic segmentation works. The goal is to segment the image into the correct classes, outlining and showing the area they occupy.

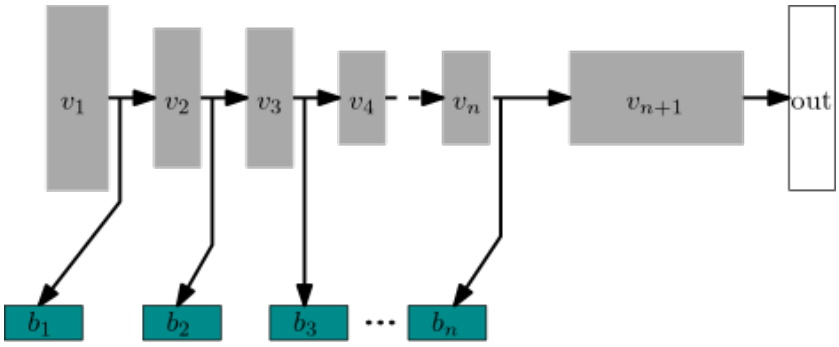
learn how to preserve the information of where each component is found in the image. Secondly, objects of interest can appear in distinct resolutions [31], meaning that a DNN trained for semantic segmentation should be capable of extracting and combining features coming from multiple resolutions.

## EE-DNNs and DNN partitioning

Thanks to advances in hardware and more availability of training data in the years 2010s, designing and training DNNs with multiple layers became possible [5]. Using models with multiple layers can enhance NN capacity to extract complex features because it can fuse information obtained in earlier layers. And with advances like residual connections [32], stability issues that come with training a DNN with hundreds of layers or more were mitigated significantly. Thus, many state-of-the-art DNNs contain a lot of layers and other computational structures, and it is a common design choice to add more layers and other resources to enhance these models' performance. Although beneficial when dealing with complex problems, deploying a DNN with many layers in a resource-constrained and latency-sensitive scenario can be difficult because the number of computations needed to reach the model's output can quickly pile up. Hence, a significant effort in paradigms like TinyML is being spent trying to simplify these models at the expense of reduced inference precision [20]. However, in many applications, simpler data can be accurately or satisfactorily classified using simpler features extracted in early layers [12, 20]. In these cases, traversing the whole DNN is unnecessary, and having a smaller NN would




(a) Regular DNN.




(b) EE-DNN.

Figure 5.2: Comparison between a DNN and an EE-DNN. In grey, we see the base DNN, which serves as the backbone of the EE-DNN. In dark cyan, the inserted early exits. The width of the $v_i$ blocks reflects the number of layers.

be desirable. Moreover, researchers observed that sometimes classes that would be correctly classified in smaller implementations can be misclassified in deep implementations, in a phenomenon referred to as "over-thinking" [20]. Thus, constructing a model that enables simpler data to be inferred using fewer layers and that resorts to all its available capacity only when necessary can offer the best of both worlds. EE-DNNs is an alternative DNN design that aims to do just that.

An EE-DNN is a multi-output DNN that modifies a regular NN architecture by inserting additional exits that are connected to intermediate layers [12, 20]. These additional exits, which are also referred to as side branches, use features extracted from the connected layers to perform an early inference attempt. If the generated output has a satisfactory level of confidence, it can be used in time-sensitive applications as auxiliary data or even as the final inference of an EE-DNN. Additionally, each exit and the layers of the base DNN (original model without early exits) connected to it make up a smaller NN. Thus, EE-DNNs offer a framework where we shift from searching from a single efficient NN to process all data to an approach that aims to construct "multiple NNs" that can minimize computations for the majority of inputs [20]. Figure 5.2 shows a comparison between an EE-DNN with a regular DNN. In dark cyan, we can see the early exits that are attached to a base DNN (in gray), connected to early layers from where they will receive feature maps they will use in their inference attempts.

Storing large DNNs resource-rich servers, such as the cloud, is another common

approach to execute DNN-based applications on resource-constrained devices. However, this approach can be problematic with latency-sensitive applications because communicating with the cloud can add significant delays. Moreover, constantly transferring high volumes of data can be power-demanding. In an EE-DNN, data flow to an early exit forms a smaller NN, if we ignore the other exits. DNN partitioning is an NN implementation approach that exploits this characteristic by distributing layers between edge and cloud instances, bringing parts of a model closer to end devices [13]. In scenarios where a base DNN is resource-demanding, layers connected to early exits can be placed in local or edge devices to offer a lightweight and latency-efficient NN. Now, transferring data to the cloud is restricted solely to samples that are difficult to classify, cases in which the whole base DNN is needed. Returning to Figure 5.2(b), when using DNN partitioning, layers that come before $b_n$ can be distributed among different instances close to end devices to offer a computationally efficient and faster inference, whereas the base DNN layers associated with $v_{n+1}$ are stored in the cloud.

## Early-exit semantic segmentation

Usually, images are high-dimensional data. DNNs that deal with this type of data tend to require high quantities of computational resources. Additionally, a common design decision is to increase the number of convolutional filters in layers close to the model's exit, enhancing the diversity of extracted features and enabling more combinations of features from earlier layers. However, in image classification, data dimensionality tends to decrease, which can prevent these computations from dramatically increasing. But problems like semantic segmentation may be harmed by severe dimensionality reduction [10], so they usually work with high-dimensional data throughout all DNN layers. Hence, it can be hard to employ a DNN trained for semantic segmentation in resource-constrained devices. Further, delays involved with transferring images to the cloud can be harmful to latency-sensitive applications, and running DNNs in non-cutting-edge equipment can introduce processing latency that can compromise these applications. For these reasons, EE-DNN together with DNN partitioning can be key components when offering semantic segmentation and other CV problems that share similar problems with resource-constrained and latency-sensitive applications.

Constructing EE-DNNs from CNNs can be difficult [20]. In these NNs, early layers usually work with very high-dimensional data, resulting in extremely high-dimensional classifiers [20]. EE-DNNs are very successful in image classification, as can be seen in the numerous research results [13, 22]. Thus, it is only natural to try to replicate this success with other CV problems. In particular, semantic segmentation

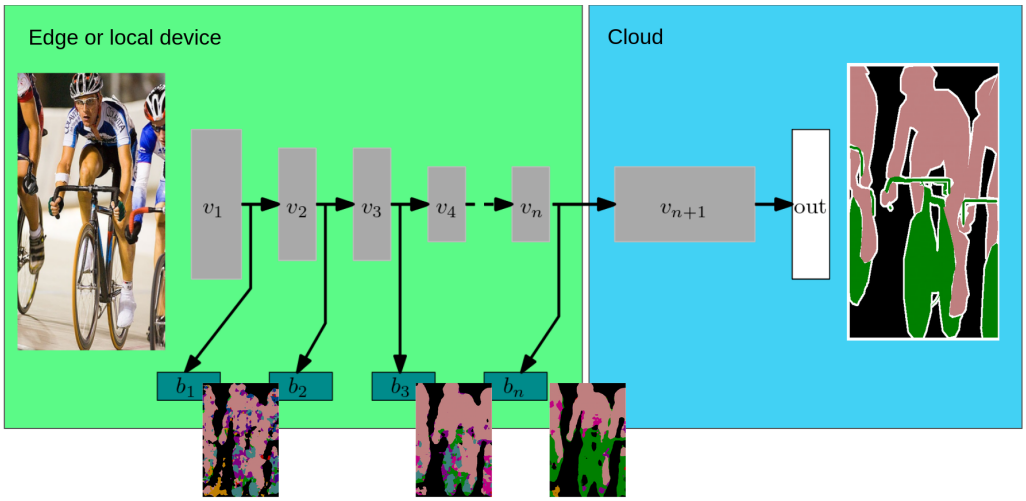



Figure 5.3: Proposed EE-DNN for semantic segmentation, showing early exits' outputs and an illustration of how this DNN can be partitioned. The early exits are the results of the proposed model, to be seen in the remainder of this chapter.

poses new challenges to EE-DNN design. Different from typical image classification, preserving localized information is important to semantic segmentation, and so is the extraction and fusion of features in multiple resolutions. Additionally, the early exits in EE-DNNs designed for image classification usually use strong dimensionality reduction procedures such as pooling and other downsampling operations [20]. These procedures don't work well with semantic segmentation, as they can cause loss of feature resolution and destroy localized information [10].

Figure 5.3 brings a sketch of the proposed EE-DNN. The idea is to address the design difficulties that semantic segmentation imposes, showing that the advantages of early exits for semantic segmentation can go beyond the production of a faster "final answer". Different from image classification, which assign a single or multiple labels to a whole picture, an EE-DNN trained for semantic segmentation can produce a crude segmentation that is detailed enough for time-sensitive applications. Figure 5.3, which brings images generated from the proposed EE-DNN, shows that early exits are capable of delivering an inference that can show the main elements of the inserted figure, albeit with a poorer quality in relation to the ground truth.

## 5.2 An EE-DNN for semantic segmentation

The experiments aim to investigate the feasibility of designing EE-DNNs for semantic segmentation. Because this CV problem needs the preservation of characteristics that can be different than image classification, some EE-DNN design choices used in the latter scenario are not suited for semantic segmentation. In the following experiments, side branches are added to a pre-trained DNN with multiple layers, to

emulate a real scenario in which we want to partition a working DNN among different instances (*e.g.* edge, cloud). The results show that early-exit semantic segmentation is possible, showing design choices that should be considered when constructing an EE-DNN for this CV application. Additionally, they demonstrate the usefulness of early-exit semantic segmentation for applications that require low-latency responses and resource-efficient computations, and manners to determine when the inference process should stop, *i.e.*, when an image can exit on a side branch.

### 5.2.1 Experiment setup

Experiments are performed using the `Pytorch` [46] framework and are available at `GitHub` [47]. The EE-DNNs were trained using the *PASCAL VOC 2012* semantic segmentation dataset [45], a dataset compiled for a computer vision competition, in which many DNN-based solutions are state-of-the-art. It consists of 2913 coloured images with 6929 identifiable objects that belong to 20 distinct labelled classes, such as people, some groups of animals, vehicles, and indoor objects. These images are split into two approximately equal-sized sets of images, by the decision of the dataset administrators. In the experiments, one of these sets is used as the training set, whereas the other is split into 60% and 40% for testing and validation sets, respectively. This gives a 50 : 20 : 30 division of samples for training, validation, and testing, respectively.

All training images are scaled to have a size equal to $256 \times 256$. To increase training sample diversity, each training image can be modified in a manner that its brightness, saturation, contrast, and hue can be altered [a]. This modification is applied solely to the original image, and the model has to learn how to deal with this corruption to generate the correct segmentation. This can aid in training by generating new data not present in the original dataset while preserving the main features of images that we want the models to learn. During each training iteration, the samples are grouped in batches of 10 image and fed into the model's input layer.

Training uses the Lovász-Softmax loss function to train EE-DNNs, an extension of the regular Mean Intersection over Union (mIoU) [48]. mIoU is the metric of choice of most semantic segmentation papers and is defined as

$$mIoU(Y, \hat{Y}) = \frac{1}{|C|} \sum_{c \in C} \frac{Y_c \cap \hat{Y}_c}{Y_c \cup \hat{Y}_c} \tag{5.1}$$

$$= \frac{1}{|C|} \sum_{c \in C} \frac{TP_c}{TP_c + FP_c + FN_c}, \tag{5.2}$$

[a] `https://pytorch.org/vision/main/generated/torchvision.transforms.ColorJitter.html`

where $Y_c$ and $\hat{Y}_c$ are the images containing the ground-truth and guesses of class $c$ ($TP$ and $FP$ are the true and false positives, $FN$ is the false negative). Essentially, it averages out the sum of IoU of each class, meaning that it can penalize models that are "specialists" in a few classes. Additionally, SGD optimizer is used with momentum equal to 0.9 and weight decay $5.10^{-}4$, and adopt

$$lr_k = lr_{base}\left(1 - \frac{k}{max\_.iter}\right)^p$$

as LR scheduling, similar to the one adopted in Berman et al. [48]. Parameter $p = 0.9$ and $k$ is the epoch's index. After preliminary analysis, the starting LR of each EE-DNN components are $lr_{base} = 2.5 \cdot 10^{-4}$ for the branches, $lr_{base} = 2.5 \cdot 10^{-6}$ for the base DNN, and $lr_{base} = 2.75 \cdot 10^{-4}$ for the original output layer. After each epoch, the trained model is evaluated with the validation set, and the best weight configuration, *i.e.*, the one that obtained the best results in the validation set, is saved as the final model. In the following experiments, EE-DNNs by computing the average of the mIoU of all exits, which includes all side branches and the base DNN original output layer.

### 5.2.2 EE-DNN Models

The base DNN used in the experiments is a pre-trained DeepLabV3, available with `torchvision` [49] module. Specifically, it consists of a modified ResNet-101, which is a backbone of many CV-oriented DNNs that is 101 layers deep [32]. The inserted branches are similar in structure to the output of the regular DeepLabV3 network, meaning that they also employ ASPP layers, Hence, even at high resolutions, we can take advantage of the atrous convolution to extract the needed low-resolution features. However, layer positioning is not as straightforward as in the typical image classification because the side branches impact the intermediate feature maps. For instance, in early experiments, a significant performance loss was observed at DeepLabV3's output layer when making its parameters equal to the pre-trained model. To this, we can attribute the fact that the branches modify the layers that preceded it to ease the features' localization, thus impacting the feature maps of the remainder of the backbone DNN.

Models are constructed with 3, 5, 7 or 9 early exits. As will be seen in the experiment results, the number and the proximity of exits can be crucial for training EE-DNNs for semantic segmentation, even if some side branches can be removed after training completion. These exits are inserted at approximately the same distance in terms of FLOPs. By measuring FLOPs we can have a rough estimate of both energy consumption and inference time [37]. So, if we can reduce FLOP, we can

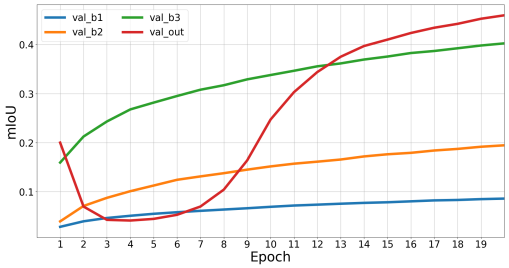
expect these two requirements to decrease. Considering that no branch is stored in the cloud, which will store the remaining layers leading to the model's output (see Figure 5.3), early exits can be split between the local device and the edge based on the requirements of applications and devices. For example, when dealing with resource-constrained devices we can add more branches in the edge, whereas we can opt to place more branches in local devices when dealing with time-sensitive applications. Additionally, in order to reduce the number of FLOPs of the overall model, some layers can be discarded after the training is complete. As the results will show, layers that are close together tend to have similar performance and can be considered redundant in the proposed EE-DNN. Finally, because DeepLabV3 is used as base DNN, the models are referred to as Branchy DeepLabV3 (BranDeepLabV3).

## 5.3 First experiments

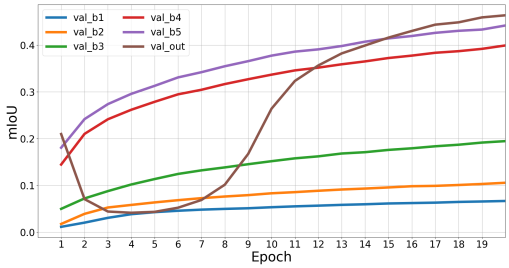
The first set of experiments is concerned with checking the viability of constructing an EE-DNN for semantic segmentation, addressing its difficulties and analysing possible applications. It starts observing how adding and training side branches impact the base DNN parameters in a manner that can affect the original output. Next, the mIoU of each exit and the number of FLOPs needed to generate each segmentation are computed. These values are obtained using $256 \times 256$ images, the same size used in the training process, These values enable us to assess the trade-offs between inferring in an early exit or going all the way towards the final output layer. Finally, a qualitative analysis using some images shows that early exits can be useful in delivering coarse segmentation that can assist low-latency applications, even when these exits have low mIoU. These final results show that we can identify the main features or some classes in these crudely segmented images.

### 5.3.1 Difficulties in training

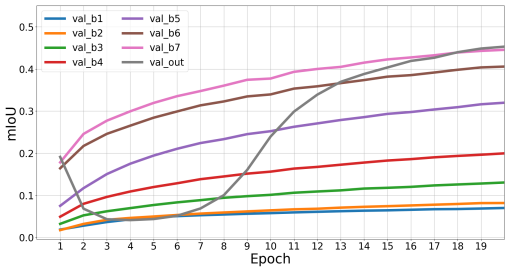
As discussed when presenting the LR parameters, the early exits impact the performance of exits that come after it. In earlier experiments, this impacted the performance of the EE-DNN final exit negatively, which was the main reason why it was decided to adopt a slightly $lr_{base}$ for this exit. To illustrate this problem, Figure 5.4 shows how the auxiliary exits still impact the original output layer at the beginning of training, even after this LR changes. Recalling that the BranDeepLabV3 is built on top of a pre-trained model, we can see a behaviour in which we start with a dip in the last exit's performance until it stabilizes (around epoch 5) and starts improving again. Although fluctuations in performance are expected, this dramatic fall can be caused by the changes promoted by the early exits. Because semantic segmentation



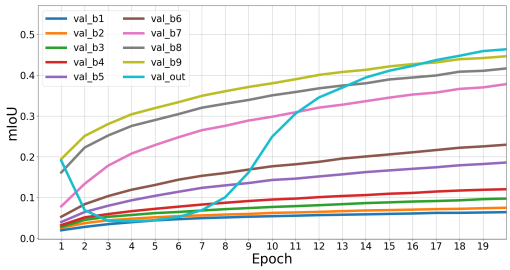

(a) Exits of the BranDeepLabV3-3.




(b) Exits of the BranDeepLabV3-5.




(c) Exits of the BranDeepLabV3-7.




(d) Exits of the BranDeepLabV3-9.

Figure 5.4: Early results on the validation set showing how the additional exits worsen the original outputs performance at the beginning of the training process. This suggests that the parameter modifications they promote impact the NN output layer.

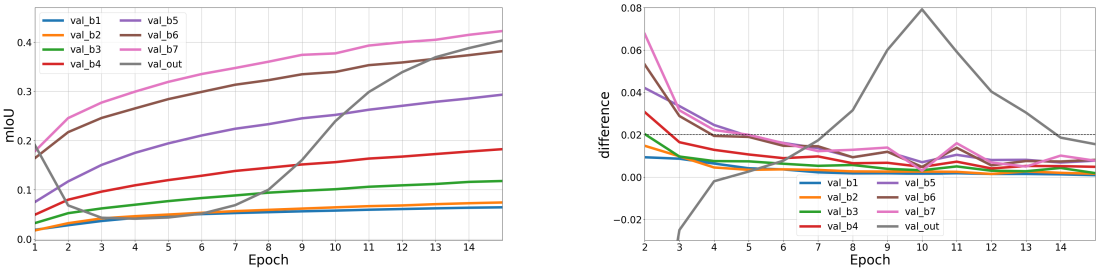
Table 5.1: BranDeepLabV3-3 performance

| | b1 | b2 | b3 | out |
|---|---|---|---|---|
| mIoU | 0.255 | 0.527 | 0.662 | 0.701 |
| FLOPs (G) | 20.49 | 30.83 | 51.44 | 60.60 |

requires localized information, it is more sensitive to feature map changes. Likely, significant parameter changes occur at the beginning of the training process, and as the training progresses these changes become more subtle. For instance, notice in Figure 5.5 how the output layer's performance starts improving after epoch 5, when the early exits exhibit a mIoU difference below 0.02 between consecutive epochs. These parameter changes impact intermediate feature maps, meaning that layers in later stages must learn how to adapt their parameters to handle these changes. Once changes become more subtle, this adaptability process became more simple. However, although a promising explanation, further investigation is needed to determine if other factors are producing these phenomena.

### 5.3.2 mIoU results

The pre-trained DeepLabV3 has $mIoU = 0.707$ in the test set, before training. In the experiments, we first explore how varying the number of insertion impacts the quality of the trained branches and the DeepLabV3 original output performance,




(a) mIoU progression.

(b) mIoU differences.

Figure 5.5: Early results BranDeepLabV3-7 on the validation set. Notice that after the consecutive epochs exhibit a mIoU difference below 0.02 in all early exits, the original output's performance starts improving.

Table 5.2: BranDeepLabV3-5 performance

| | b1 | b2 | b3 | b4 | b5 | out |
|---|---|---|---|---|---|---|
| mIoU | 0.201 | 0.340 | 0.528 | 0.666 | 0.686 | 0.703 |
| FLOPs (G) | 17.04 | 23.94 | 30.83 | 51.44 | 56.02 | 60.60 |

given the problems exposed in the previous subsection. Tables 5.1-5.4 show the mIoU (Equation 5.2) of each exit and the number of FLOPs needed to reach each output. FLOPs were computed using a $256 \times 256$ RGB picture, and we ignore the FLOP computation of the branches that preceded it. This is done because, as discussed in the model conceptualization, it is assumed that some branches can be discarded after training.

As anticipated, a progressive improvement of mIoU is observable as we move from the first early exit towards the BranDeepLabV3 final output. Furthermore, two additional conclusions can be drawn from these results. First, regardless of the model's number of branches, exits whose positions are close in terms of FLOPs tend to have the same mIoU. For example, $b1$ in BranDeepLabV3-3 and $b2$ in BranDeepLabV3-7 have the same FLOP (approximately $20.49G$ FLOPs) and nearly identical mIoU, with the former having $mIoU_{b3} \approx 0.255$ and the latter having $mIoU_{b3} \approx 0.259$. This suggests that the branch position influences the maximum mIoU achievable. However, having more branches means that a designer has more freedom to choose which exits to discard when deploying the trained model. Additionally, mIoU averages Intersection over Union (IoU) across all classes, meaning that it is possible to train these branches to specialize in easier or task-sensitive classes (e.g., identifying people on autonomous driving).

Another result is that it seems that adding more branches to the base model improves both the performance of the original output and the performance of the branches closer to it. Notice that $mIoU_{out}$ progressively improves as the number of early exits increases. Moreover, this performance improvement is more pronounced with the branch inserted at around $51.44G$ FLOPs, which starts with a mIoU score

Table 5.3: BranDeepLabV3-7 performance

| | b1 | b2 | b3 | b4 |
|---|---|---|---|---|
| mIoU | 0.193 | 0.259 | 0.420 | 0.528 |
| FLOPs (G) | 14.74 | 20.49 | 26.24 | 30.83 |

| | b5 | b6 | b7 | out |
|---|---|---|---|---|
| mIoU | 0.631 | 0.674 | 0.689 | 0.705 |
| FLOPs (G) | 36.58 | 51.44 | 56.02 | 60.60 |

Table 5.4: BranDeepLabV3-9 performance

| | b1 | b2 | b3 | b4 | b5 |
|---|---|---|---|---|---|
| mIoU | 0.176 | 0.206 | 0.290 | 0.398 | 0.514 |
| FLOPs (G) | 13.59 | 17.04 | 21.64 | 25.09 | 29.68 |

| | b6 | b7 | b8 | b9 | out |
|---|---|---|---|---|---|
| mIoU | 0.562 | 0.655 | 0.674 | 0.688 | 0.707 |
| FLOPs (G) | 33.13 | 37.73 | 51.44 | 56.02 | 60.60 |

of 0.662 in BranDeepLabV3-3 but can reach an mIoU of 0.674 in BranDeepLabV3-9 inserted branches. As was discussed previously, each early exit needs the feature maps it receives from the base DNN to carry positional information. Hence, any modifications that occur in the layers that precede it during the training process can impact the next branches (and the base model's output) negatively. These results, specifically the ones found with the deepest implementations, suggest that adjacent exits can help one another during training. However, further investigation is needed to reach a definitive conclusion.

### 5.3.3 Qualitative Results

Figure 5.6 shows some output images generated by BranDeepLabV3-7. Even though the segmentations in the early stages are crude, for some applications they can be detailed enough. For example, in autonomous driving, $b2$ or $b3$ outputs (Figures5.6(b) and 5.6(c)) might be good enough to identify a cyclist, enabling the agent to take a quick response. Hence, even if it would be beneficial to have a finer segmentation (produced at a cloud, for example), the cruder early exit can be used in latency-sensitive applications to initiate an action. Differently, to identify the biker in Figure 5.6(p) on a similar application, $b5$ output seems more appropriate. However, if we are interested in the region occupied by an object, Figure 5.6(r) can be useful. Altogether, these images show how the quality of generated segmentations can vary from picture to picture, showing how the mIoU results should not be considered alone. Additionally, there is the possibility of fine-tuning early exit to identify sensitive classes. Again, returning to the autonomous driving example, to

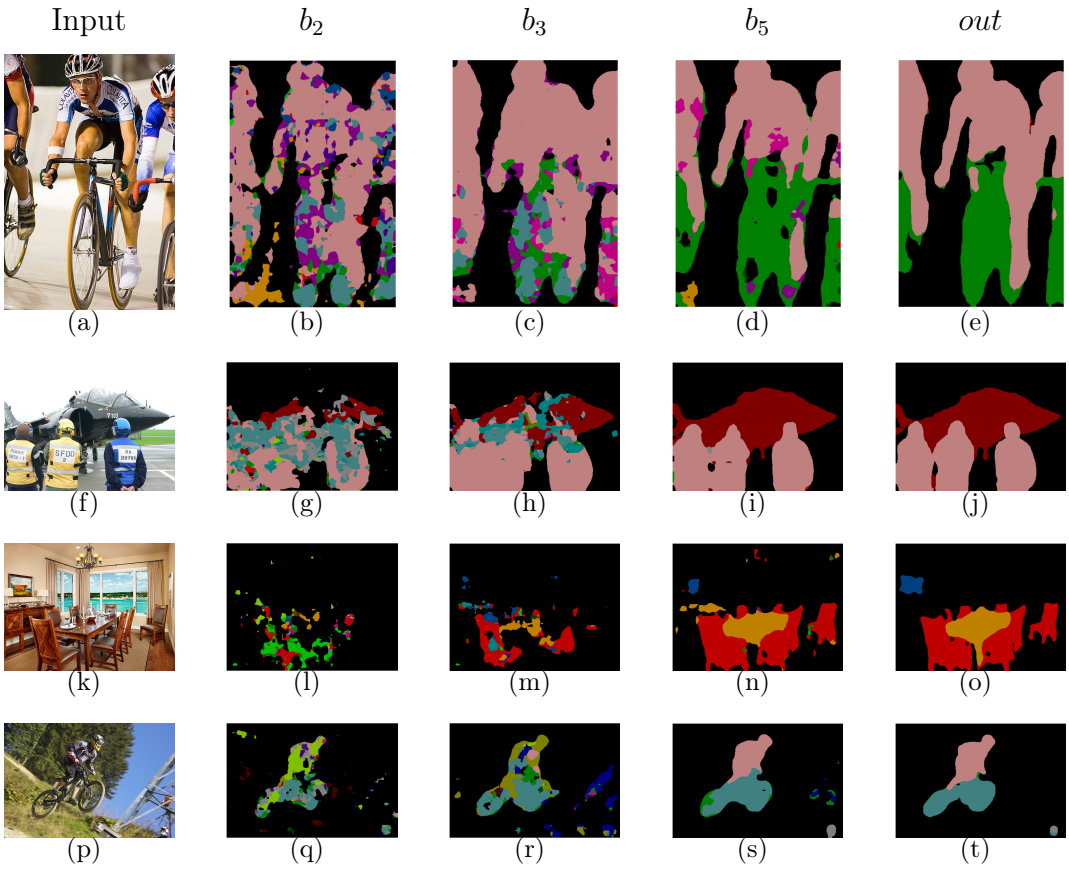



Figure 5.6: Segmented images from BranDeepLabV3-7. Colour coding can be found in Appendix 5.A.

minimize collisions with humans, we can perform training that encourages the side branches to identify vulnerable road users. Differently, in other applications, early exists can be trained to differentiate end users from possible obstacles, assisting time-sensitive proactive applications in computer-vision assisted communication [7].

Another result brought by Figure 5.6 is how outputs can vary between different exits. For instance, exits closer to the input, such as $b2$ and $b3$, can oversize some classes, as evidenced by the contours of people in segmentations generated from Figures 5.6(a) and 5.6(f). However, these early exits may also misclassify objects, even if the object shape is correctly identified, as seen in Figure 5.6(p), or fail to provide any meaningful information, as in Figure 5.6(l). Comparing these results with segmentations generated by $b5$ and *out*, we can observe a refinement of segmented information, with cluster size reduced in cases of oversizing and the removal of non-existent classes. As will be discussed later, these behaviours should be considered when devising metrics to measure if data can exit the EE-DNN.

Figure 5.7 brings the relationship between image size and the number of FLOP needed to reach each exit of BranDeepLabV3-7, showing in black the number of FLOP needed to produce each exit in Figure 5.6(b)-5.6(e). Images can appear in

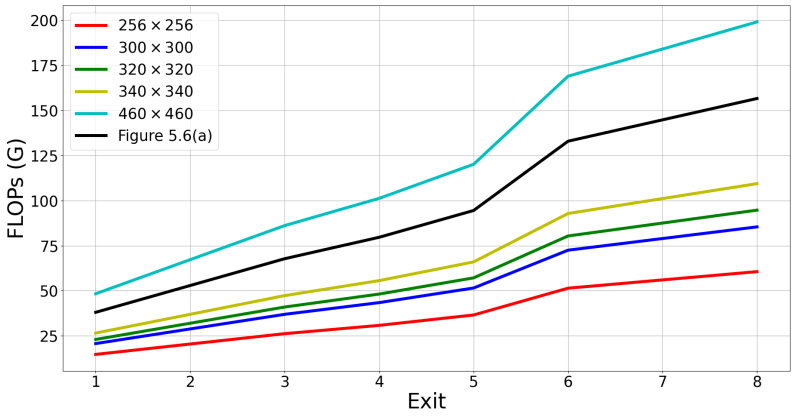



Figure 5.7: Number of FLOPs of each exit of the BranDeepLabV3 with 7 early exits.

many shapes, meaning the number of computations and expended energy can vary a lot from image to image. It is noticeable that the size of the image is an issue that can worsen both latency response and energy consumption, as the later layers are the ones where the increase of FLOP is more pronounced. Thus, it is evident that early exits are a valuable tool for time-sensitive and resource-constrained implementations.

## 5.4 When should inference stop?

Determining when to stop the inference process in a semantic segmentation application can be challenging because a DNN produces images as output. When working with image classification, many works resort to the output probabilities generated at each exit, and the inference process stops whenever a probability exceeds a threshold value [13, 22]. In semantic segmentation, we deal with pixel-level classification, thus using this approach can be intractable, especially when working with high-dimensional pictures. Additionally, this CV problem also involves clustering, meaning that analysing each pixel separately can lead to poor performance. Hence, the success of EE-DNN in multi-stage inference scenarios in which we want to avoid traversing all the base DNN layers rely on devising a good exit criterion that takes all this into account.

A possible exit criterion is comparing the exits of two consecutive side branches to check if there are any significant changes. For example, consider $\hat{Y}_i$ and $\hat{Y}_{i+1}$ as output images of the $i$-th and $i+1$-th early exits. If $\hat{Y}_i$ is close to the true segmentation, then $\hat{Y}_{i+1}$ will likely have the same segmentations, albeit with a better quality. To do this, we can use a metric to compare the difference between these two segmented images and, much like in the image classification case, stop the inference process if it overcomes a predefined threshold. If the two images are similar, we can consider $\hat{Y}_{i+1}$ as the EE-DNN inferred segmentation.

The following experiments investigates Variation of Information (VI) [50] as a possible exit metric. VI measures the information exchange when we change clusters, being a criterion for comparing partitions [50]. Supposing that $X$ and $Y$ represent two images we want to compare,

$$VI(X, Y) = H[X|Y] + H[Y|X], \tag{5.3}$$

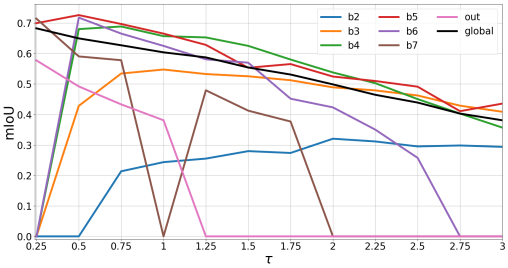
meaning that VI is a sum of the conditional entropies of $X$ given $Y$ and $Y$ given $X$. In the proposed implementation, when we want to check if a $\hat{Y}_i$ can be the EE-DNN exit, we have $X = \hat{Y}_{i-1}$ and $Y = \hat{Y}_i$. Additionally, each conditional entropy is capturing different segmentation errors. If the segmented region is correct but $X$ contains some inexistent classes, $H[X|Y]$ can be seen as measuring if $Y$ removed or reduced any wrong class from the segmentation generated in $X$. On the other hand, $H[Y|X]$ can indicate if the area occupied by the correct classes started reflecting the correct segmentation [b]. Hence, there is an opportunity to exploit them according to the nature of errors generated by each early exit. To investigate this, the last experiments of this section make a preliminary study on the importance of these entropies individually.

In the experiments, the threshold value ($\tau$) can vary between 3 and 0.25, representing scenarios where exiting the EE-DNN is either easier or more stringent. Additionally, the conditional entropies are measured ignoring the background class, meaning that the following results (and the analysis in Appendix 5.B) use this assumption. The experiments utilize BranDeepLabV3s with 7 and 9 early exits trained in previous experiments. When an image comparison surpasses the threshold value, indicating successful early segmentation, the deviation from the ground truth is measured. Ultimately, we obtain an mIoU for each exit, a global mIoU reflecting the overall model performance, and the percentage of output generation.

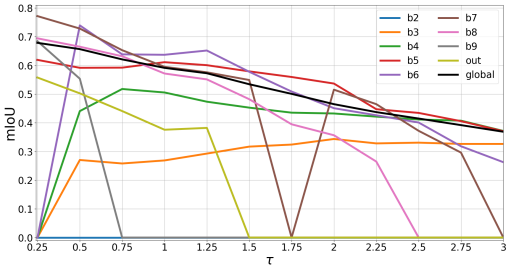
### 5.4.1 VI results

Figure 5.8 brings the results using VI as the exit metric. Both Figure 5.8(c) and 5.8(d) show that when $\tau = 3$ about 83% images are segmented in the two first early exits of BranDeepLabV3-7, and about 86% on or before $b4$ in BranDeepLabV3-9. This is non-ideal, as these exits aren't suited to classify every image, given that none has a $mIoU \geq 0.45$ (as can be seen in Tables 5.3 and 5.4). This compromised the performance of the model as a whole, as both BranDeepLabV3-7 and 9 wasn't able to reach $mIoU = 0.4$. Things start to improve when $\tau = 1.5$. In the case of the model with 7 early exits, global $mIoU \approx 0.55$ with about 0.45% images being segmented

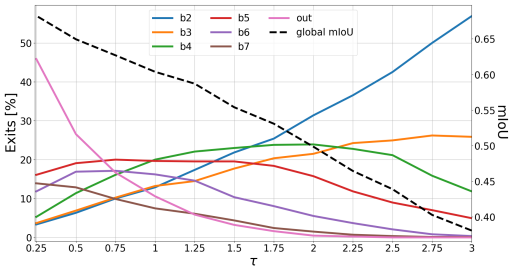
[b] For more details on how we can leverage these two conditional entropies in the following experiments see Appendix 5.B



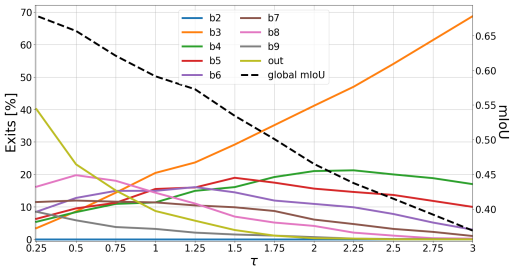

(a) mIoU of BranDeepLabV3-7.




(b) mIoU of BranDeepLabV3-9.




(c) Percentage of images that came out through each exit of BranDeepLabV3-7.

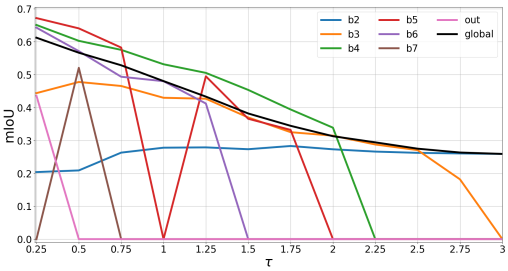



(d) Percentage of images that came out through each exit of BranDeepLabV3-9.

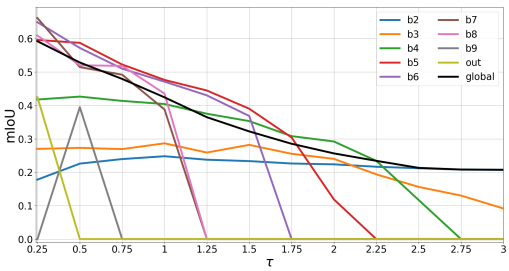
Figure 5.8: VI-based exit criterion results. Notice that when $\tau = 1.0$, BranDeepLabV3-7 can reach a global $mIoU \approx 0.6$ with more than 60% of images exiting on and before $b5$. Similarly, for the same $\tau$, BranDeepLabV3-9 has about 60% exits occurring before $b6$, with global $mIoU$ slightly smaller than 0.6.

in $b4$ or earlier branches. Remembering that $mIoU_{out} = 0.705$, this indicates that at $\tau = 1.5$ it was possible to reach 80% of the last output's performance. However, notice that this $\tau$ is still too loose to force the hardest images (that can be classified by DeepLabV3) to reach the last exit, as less than 5% reached it. As can be seen in Figure 5.8(a), notice that we can exploit $b_3$ and $b_4$ to the fullest as $mIoU_{b_3} \approx 0.53$ and $mIoU_{b_4} \approx 0.62$, whereas $mIoU_{out} = 0$. A similar pattern can be seen with BranDeepLabV3-9, with $mIoU_{out} \approx 0.53$, $mIoU_{b4} \approx 0.45$, $mIoU_{b5} \approx 0.58$, and 65% of inference occurring on and before $b5$.

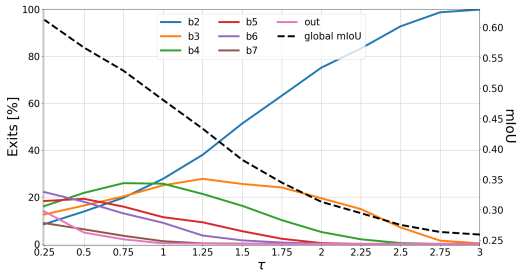
When $\tau = 1$, BranDeepLabV3-7 is able to reach $mIoU \approx 0.6$, reaching about 85% of $mIoU_{out}$ (Table 5.3), with approximately 46% exits occurring on of before $b4$ and 66% occurring on or before $b5$. Notice that at this $mIoU_{b_3} \approx 0.55$, $mIoU_{b4} \approx 0.66$, and $mIoU_5 \approx 0.67$. In the case of BranDeepLabV3-9, $mIoU_{global} > 0.6$ is reached after $\tau = 0.75$. This is likely due to the higher number and density of early exits. We can suppose that there are small changes between consecutive exits, so we need a small $\tau$ to account for this. As discussed in the previous section, some branches can be discarded after training. As seen in Table 5.4, close exits had similar $mIoU$, which means they likely generate similar segmentations.



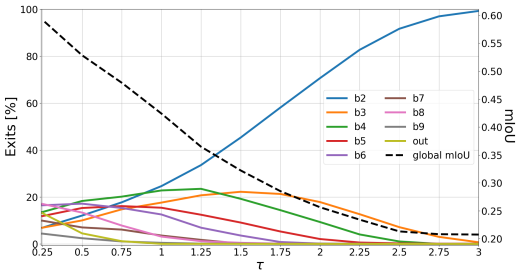

(a) mIoU of BranDeepLabV3-7.




(b) mIoU of BranDeepLabV3-9.




(c) Percentage of images that came out through each exit of BranDeepLabV3-7.




(d) Percentage of images that came out through each exit of BranDeepLabV3-9.

Figure 5.9: Using $H[X|Y]$ as exit criterion. Because we don't have the influence of $H[Y|X]$, more than 50% of the images exit both EE-DNNs in the earlier layers when $\tau \geq 1$. Additionally, $mIoU \leq 0.6$ in almost all cases.

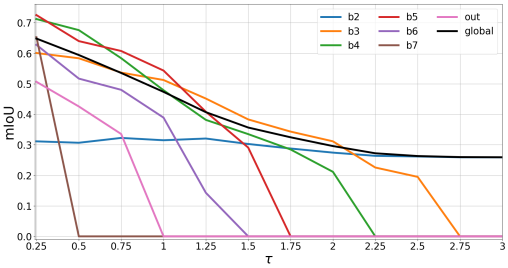
### 5.4.2 Measuring $H[X|Y]$

Recalling equation 5.3, $V(X, Y)$ is given as a sum of two conditional entropies that measure the clustering differences between two images. Remembering that $X = \hat{Y}_{i-1}$ and $Y = \hat{Y}_i$ in the proposed implementation, $H[\hat{Y}_{i-1}|\hat{Y}_i]$ measures if $\hat{Y}_i$ has the correct cluster regions and if it removed any non-existant class. When this entropy is small, we can assume that the classes are starting to appear in the correct regions. The results in Figure 5.9 show that this metric encourages more exits in early branches, with more than 50% of the images exiting both EE-DNNs in the earlier layers when $\tau \geq 1$. However, when focusing on performance, in all situations except BranDeepLabV3-7 when $\tau = 0.25$ $mIoU_{global} < 0.6$. This suggests that, at least for the first early exits, we have to take into consideration if the shapes of the regions of interest (which are better captured by $H[Y|X]$) are correct.

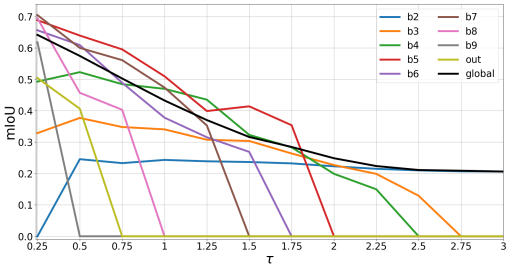
### 5.4.3 Measuring $H[Y|X]$

Different from the previous case, when we compute $H[\hat{Y}_i|\hat{Y}_{i-1}]$ we assume that we have the correct classes and are checking if $\hat{Y}_i$ is closer to the appropriate shape [c]). When compared with $H[X|Y]$, $H[Y|X]$ seems to play a more significant role when comparing the outputs of the first early exits, as can be seen with their

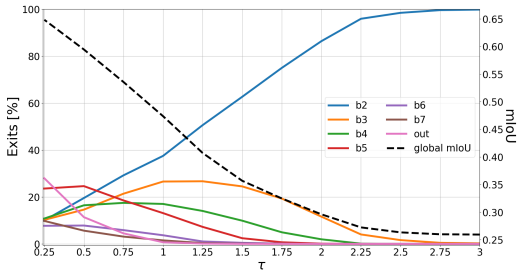
[c] See Appendix 5.B



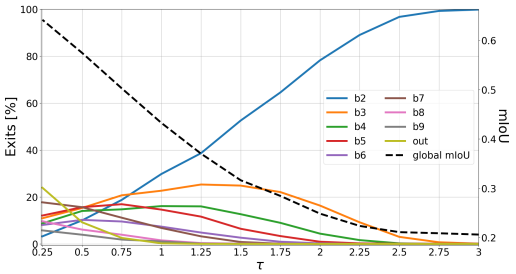

(a) mIoU of BranDeepLabV3-7.




(b) mIoU of BranDeepLabV3-9.




(c) Percentage of images that came out through each exit of BranDeepLabV3-7.




(d) Percentage of images that came out through each exit of BranDeepLabV3-9.

Figure 5.10: Using $H[Y|X]$ as exit criterion. Compared to the $H[X|Y]$, this conditional entropy seems more appropriate for the first, given that it has $mIoU_{global}$ better results.

results in the two stricter cases ($\tau = 0.25$ and $\tau = 0.5$). For example, analysing BranDeepLabV3-7 when $\tau = 0.5$, approximately 15% images are segmented in $b3$, but $mIoU_{b3,H[X|Y]} \approx 0.48$ and $mIoU_{b3,H[Y|X]} \approx 0.58$. Similarly, in BranDeepLabV3-9 (for the same $\tau$), $mIoU_{b4,H[X|Y]} \approx 0.43$ with approximately 18% of images segmented, and $mIoU_{b4,H[Y|X]} \approx 0.52$ with approximately 14% of images segmented. Remembering the qualitative analysis, the first segmentations can have fewer clusterings than the ones generated in later exits, which can explain why we need a metric that can measure new "additions". Moreover, when the classes are close to their correct position, they usually have the wrong size. However, we must take into consideration that different types of segmentation errors can occur in different parts of the EE-DNN, so learning how to balance the importance of these two components of VI to reflect when one condition is more predominant than the other can be key for future performance improvements.

## 5.5 Conclusion

Given the success of EE-DNNs in image classification, it is only natural to try to replicate it in other CV problems, like semantic segmentation. Many time-sensitive applications that require semantic segmentation can benefit from early attempts at segmentation. Additionally, when we combine these multi-exit models with DNN

partitioning, we can offer smaller and resource-efficient NNs that can be placed in resource-constrained devices and that can infer the majority of input data. The first experiments present a new implementation suited for early-exit semantic segmentation, showing the usefulness of the approach. The results show that the earlier exits' FLOPs, which allows us to estimate latency and energy consumption, increase at a slower pace than the later stages, and that they are capable of delivering coarse segmentations that can outline relevant features. For instance, we show that in earlier stages, where mIoU is below 0.4, the early exits are capable of delivering a segmentation displaying relevant features. This result, in particular, indicates an opportunity to train the early exits to distinguish sensitive classes, to be investigated in future research.

The second set of experiments investigates VI as a possible exit metric, and provides an initial analysis on its components as a mean to capture distinct segmentation errors. The results showed that VI can be a valuable metric to determine when a segmented image can leave the EE-DNN. When using VI, which gives equal importance to the two conditional entropies that make up this metric, we see that the proposed EE-DNN can achieve 85% of $mIoU_{out}$ performance (Table 5.3) with 60% of images exiting on or before the fifth early exit. Additionally, although the results measuring the importance of the conditional entropies are preliminary, they can be a first step in understanding how we can modify VI so it can reflect the type of errors occurring in different parts of the EE-DNN. In the future, further experiments to broaden our understanding on the nature of the segmentation errors and in which parts of the EE-DNN they occur can help us modify VI to improve exit evaluation. Moereover, it is also interesting to evaluate if removing early exits and changing the reference side branch (fixed to $b1$ in these experiments) can also improve performance.

## 5.A Pascal Voc colour scheme

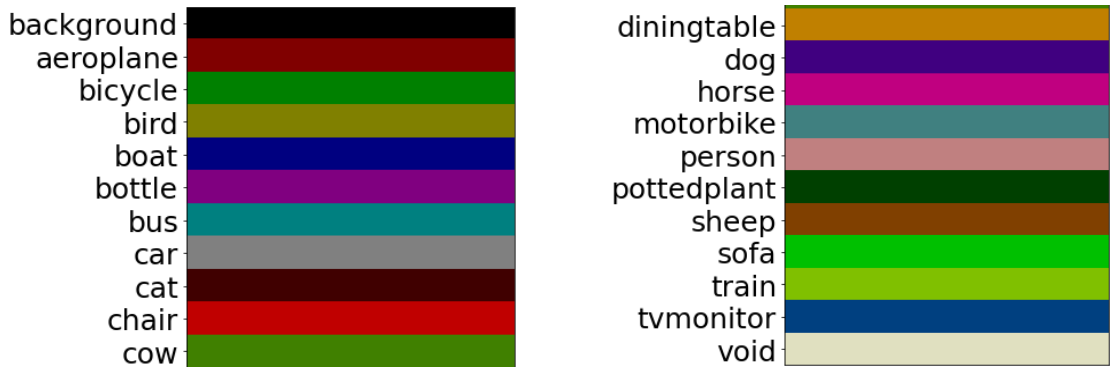
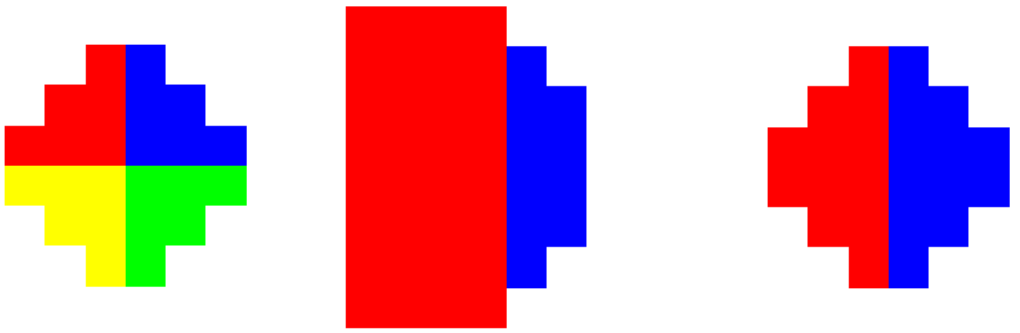



Figure 5.11: Labelled classes of Pascal Voc, along with their respective colour coding.

## 5.B A visual interpretation of VI *



(a) $X_1$ (b) $X_2$ (c) $Y$

Figure 5.12: Visual interpretation of VI in the studied scenario. Going from $X_1$ to $Y$ illustrates a situation where classes are removed and $X_2$ to $Y$ a situation where the area of a segmented regions are corrected. Figures have size $8 \times 8$ pixels.

In the experiments we use VI as a metric to decide when a segmented image can exit the EE-DNN. Recaling equation 5.3, it is assumed that $X = \hat{Y}_{i-1}$ and $Y = \hat{Y}_i$, *i.e.*, $Y$ is the output of the early exit we are computing VI and $X$ the output of the previous exit. If we suppose that the previous output is Figure 5.12(a) and the current output is Figure 5.12(c), we can see that some segmented regions where removed. According to VI definition, $H[Y|X_1] = 0$ given that the clusters in $Y$ can be obtained by merging the clusters in $X_1$ [50]. Indeed this is the case, as

$$H[X_1|Y] \approx 1.00 \text{ and } \quad H[Y|X_1] = 0,$$

meaning that $H[X|Y]$ can capture when inesistent classes, *i.e.*, wrong guesses from early exits stop appearing. Conversily, in situations that segmented regions are correct but have the wrong size, like when we compare Figure 5.12(b) with Figure 5.12(c), the second conditional entropy can be more significant. For instance, in this example,

$$H[X_2|Y] = 0 \qquad \text{and} \quad H[Y|X_2] \approx 0.73.$$

Hence, these two conditional entropies individually measure each of these segmentation errors occuring between consecutive exits. Moreover, we can leverage the likelihood of each error occurring in different parts of the EE-DNN to assign different weights to them, thereby improving performance.

*The results in this section were computed ignoring background class (as in the previous experiments). See `https://scikit-image.org/docs/stable/api/skimage.metrics.html#skimage.metrics.variation_of_information` for details.

# Chapter 6

# Conclusion

This thesis goal is to engage in the discussion of efficient implementation of DNNs in resource-constrained and latency-sensitive scenarios. DNNs can be extremely resource-demanding and require a lot of computation steps, especially configurations with multiple layers. As problems become more complex, and training DNNs with huge numbers of layers becomes more feasible, state-of-the-art models become hard to implement in the aforementioned scenarios. Even though hardware advances can ease implementation issues, costs can play a key role, and so the fact that with more resources the greater are the opportunities to design newer and more costly DNNs. Hence, having alternative NN designs whose goal is to reduce the amount of parameters to be stored in resource-constrained devices and instances that require low-latency responses is greatly appreciated. For instance, TinyML is an active research area that seeks this goal.

This thesis discusses two DNN implementation scenarios. The first, in chapter 4, discussed the problems with seeking performance improvements through the increase of DNN depth. Specifically, it analyses how AEs are useful in IoT problems, which usually involve resource-constrained devices. In this and similar scenarios, deploying a deep AE can be difficult. The proposed AAE aims to address this problem, offering an alternative design in which performance improvements through DNN depth increase can be sought without imposing more layers to be stored in IoT sensor and actuator nodes. The second, in chapter 5, discusses the need for and the usefulness of multi-output DNNs in implementing CV applications that offer fast- and resource-efficient responses. Together with DNN partitioning, these DNNs can be spread between edge and cloud instances, offering a multistage inference scenario in which only hard-to-treat data are outsourced to the cloud. In this chapter, the design of an EE-DNN capable of semantic segmentation is proposed, discussing its usefulness to applications like autonomous driving, and the feasibility of such DNN. Overall, both scenarios show how NN design can be modified to include training restriction and model features that can assist DNN implementation in scenarios

where deep multi-layered models are hard to implement.

## 6.1 Summary of thesis' results

The main results are summarised as follows. Many applications that involve data sensing and transmission in IoT networks employ AEs and usually need to store the encoding layers of these models in resource-constrained devices. The proposed AAE offers these devices an alternative AE design that can limit the number of encoding layers without significant performance losses and is even capable of outperforming its symmetric counterpart. In a one-dimensional temporal-data transmission scenario, the results in chapter 4 show that AAEs encoders require significantly fewer parameters and FLOPs per data compression than symmetric AEs with similar decoding configuration. Additionally, thanks to this smaller number of parameters, deep AAEs are shown to be easier to train, especially in experiments with noisy data. In many scenarios, the number of training examples can be limited, which can make it difficult to train deep NN configurations. Finally, CAAEs results, which were the best-performing models, showed that we can pursue further performance improvements by modifying the decoders to exploit the particularities of the phenomenon of interest. Overall, these results show that we can move away from the traditional approach of a mirrored encoder-decoder design by constructing an AE architecture that reflects our needs. In this case, a lightweight and compact encoder coupled with a more powerful decoder that can address the shortcomings of designing an AE with fewer encoding layers.

EE-DNN, together with DNN partitioning, are known to be capable of offering multi-stage image classification that can classify simpler data faster and energetically efficient. Given this success, it is only natural to try to replicate it in other CV problems. In chapter 5, an EE-DNN for semantic segmentation is constructed, showing how its design and training differs from EE-DNN designed for image classification. Additionally, the results show how these multi-output DNNs can be useful for latency-sensitive applications, as early exits can deliver coarsely segmented images that can identify the main features in an image while requiring fewer FLOPs. Finally, the last experiments show the difficulties of devising an exit strategy and the importance of similarity metrics. In it, a first look on the importance of measuring the conditional entropies between segmented images was discussed, and how we exploit them in different branches according to the nature of updates between consecutive exits. Altogether, these pioneering experiments on early-exit semantic segmentation show the importance of EE-DNN in time-sensitive and resource-efficient implementations, giving directions on how to overcome some design obstacles.

## 6.2 DNNs continuously becoming more ubiquitous

Thanks to its data-driven approach and recent advances in NN theory, DNNs are continuously being widespread in many research and industrial paradigms. DNNs are now state-of-the-art in many CV and NLP applications, and are being used to tackle a wide range of problems that involve complex and large volumes of data. For example, CV assisted communication [7] is an emerging paradigm whose objective is to incorporate CV applications into communication systems and protocols to enrich data quality and decision-making. In summary, many distinct applications will seek DNNs, each with its own restrictions and requirements. This presents an extremely heterogeneous future, and having different DNN architectures to meet each situation will be necessary.

Having DNNs that require fewer layers or that can compute the majority of data with fewer resources are fundamental to local implementations in resource-constrained scenarios and edge implementations, which aim to bring DNNs close to end-users. The design choices discussed in this thesis, together with paradigms like TinyML, can offer researchers and developers computationally efficient NNs, which can contribute to an even greater diffusion of DNN-based systems and applications in scenarios where these models are not easily accessible.

## 6.3 Future prospects

In addition to future works discussed briefly at the end of chapters 4 and 5, AAEs and early-exit semantic segmentation can be helpful to other applications. For instance, the asymmetrical design discussed with AAEs can help other DNN implementations that employ encoder-decoder architecture, like Generative Adversarial Networks (GANs) [51] and U-Nets [52]. Additionally, paradigms like knowledge distillation [24] can employ AAEs to assist techniques that aim to replace a set of layers with a smaller number, and knowledge distillation, in turn, can help train and design EE-DNNs. Furthermore, specifically concerning AE-based implementations in IoT, designing AAEs to work with multi-source data is important [23]. Now, regarding EE-DNN, many applications that seek semantic segmentation to deal with video data can combine data generated in edge and cloud instances to deliver faster and more reliable inferences at early exits. Additionally, other CV problems (and possibly other types of data) need preservation of localized and structural information, thus can work with strong downsampling and pooling operations found in traditional EE-DNNs. Hence, the observations found in the EE-DNN design for semantic segmentation can be helpful to open another direction in the multi-exit DNN design.

# References


[1] GARCIA-GARCIA, A., ORTS-ESCOLANO, S., OPREA, S., et al. "A survey on deep learning techniques for image and video semantic segmentation", *Applied Soft Computing*, v. 70, pp. 41–65, 2018.

[2] OTTER, D. W., MEDINA, J. R., KALITA, J. K. "A survey of the usages of deep learning for natural language processing", *IEEE transactions on neural networks and learning systems*, v. 32, n. 2, pp. 604–624, 2020.

[3] HAN, H., SIEBERT, J. "TinyML: A systematic review and synthesis of existing research". In: *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 269–274. IEEE, 2022.

[4] ABADADE, Y., TEMOUDEN, A., BAMOUMEN, H., et al. "A comprehensive survey on tinyml", *IEEE Access*, 2023.

[5] LECUN, Y., BENGIO, Y., HINTON, G. "Deep learning", *nature*, v. 521, n. 7553, pp. 436–444, 2015.

[6] GILBERT, M. S., PACHECO, R. G., COUTO, R. S., et al. "Unlocking Early-Exiting Semantic Segmentation with Branched Networks". In: *2023 IEEE Latin-American Conference on Communications (LATINCOM)*, pp. 1–6. IEEE, 2023.

[7] NISHIO, T., KODA, Y., PARK, J., et al. "When wireless communications meet computer vision in beyond 5G", *IEEE Communications Standards Magazine*, v. 5, n. 2, pp. 76–83, 2021.

[8] MOHAMMADI, M., AL-FUQAHA, A., SOROUR, S., et al. "Deep learning for IoT big data and streaming analytics: A survey", *IEEE Communications Surveys & Tutorials*, v. 20, n. 4, pp. 2923–2960, 2018.

[9] BOCHIE, K., GILBERT, M. S., GANTERT, L., et al. "A survey on deep learning for challenged networks: Applications and trends", *Journal of Network and Computer Applications*, p. 103213, 2021.

[10] CHEN, L.-C., PAPANDREOU, G., SCHROFF, F., et al. "Rethinking Atrous Convolution for Semantic Image Segmentation". 2017.

[11] WANG, P., CHEN, P., YUAN, Y., et al. "Understanding convolution for semantic segmentation". In: *2018 IEEE winter conference on applications of computer vision (WACV)*, pp. 1451–1460. Ieee, 2018.

[12] TEERAPITTAYANON, S., MCDANEL, B., KUNG, H.-T. "Branchynet: Fast inference via early exiting from deep neural networks". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469. IEEE, 2016.

[13] PACHECO, R. G., BOCHIE, K., GILBERT, M. S., et al. "Towards edge computing using early-exit convolutional neural networks", *Information*, v. 12, n. 10, pp. 431, 2021.

[14] CORREA, J. D. A., PINTO, A. S. R., MONTEZ, C. "Lossy data compression for iot sensors: A review", *Internet of Things*, v. 19, pp. 100516, 2022.

[15] ALSHEIKH, M. A., LIN, S., NIYATO, D., et al. "Rate-distortion balanced data compression for wireless sensor networks", *IEEE Sensors Journal*, v. 16, n. 12, pp. 5072–5083, 2016.

[16] GHOSH, A. M., GROLINGER, K. "Deep Learning: Edge-Cloud Data Analytics for IoT". In: *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, pp. 1–7. IEEE, 2019.

[17] LAAKOM, F., RAITOHARJU, J., IOSIFIDIS, A., et al. "Reducing redundancy in the bottleneck representation of the autoencoders", *arXiv preprint arXiv:2202.04629*, 2022.

[18] LEE, W.-H., OZGER, M., CHALLITA, U., et al. "Noise learning-based denoising autoencoder", *IEEE Communications Letters*, v. 25, n. 9, pp. 2983–2987, 2021.

[19] MAJUMDAR, A., TRIPATHI, A. "Asymmetric stacked autoencoder". In: *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 911–918. IEEE, 2017.

[20] SCARDAPANE, S., SCARPINITI, M., BACCARELLI, E., et al. "Why should we add early exits to neural networks?" *Cognitive Computation*, v. 12, n. 5, pp. 954–966, 2020.

[21] LO, C., SU, Y.-Y., LEE, C.-Y., et al. "A dynamic deep neural network design for efficient workload allocation in edge computing". In: *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 273–280. IEEE, 2017.

[22] PACHECO, R. G., COUTO, R. S. "Inference time optimization using branchynet partitioning". In: *2020 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–6. IEEE, 2020.

[23] AZAR, J., MAKHOUL, A., BARHAMGI, M., et al. "An energy efficient IoT data compression approach for edge machine learning", *Future Generation Computer Systems*, v. 96, pp. 168–175, 2019.

[24] GOU, J., YU, B., MAYBANK, S. J., et al. "Knowledge distillation: A survey", *International Journal of Computer Vision*, v. 129, pp. 1789–1819, 2021.

[25] O'MAHONY, N., CAMPBELL, S., CARVALHO, A., et al. "Deep learning vs. traditional computer vision". In: *Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1 1*, pp. 128–144. Springer, 2020.

[26] CHARTE, D., CHARTE, F., GARCÍA, S., et al. "A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines", *Information Fusion*, v. 44, pp. 78–96, 2018.

[27] CYBENKO, G. "Approximation by superpositions of a sigmoidal function", *Mathematics of control, signals and systems*, v. 2, n. 4, pp. 303–314, 1989.

[28] LECUN, Y., BENGIO, Y., OTHERS. "Convolutional networks for images, speech, and time series", *The handbook of brain theory and neural networks*, v. 3361, n. 10, pp. 1995, 1995.

[29] CHOLLET, F. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[30] YU, F., KOLTUN, V. "Multi-scale context aggregation by dilated convolutions", *arXiv preprint arXiv:1511.07122*, 2015.

[31] CHEN, L.-C., PAPANDREOU, G., KOKKINOS, I., et al. "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs", *IEEE transactions on pattern analysis and machine intelligence*, v. 40, n. 4, pp. 834–848, 2017.

[32] HE, K., ZHANG, X., REN, S., et al. “Deep Residual Learning for Image Recognition”. 2015.

[33] LI, S., JIAO, J., HAN, Y., et al. “Demystifying resnet”, *arXiv preprint arXiv:1611.01186*, 2016.

[34] TILAK, S., ABU-GHAZALEH, N. B., HEINZELMAN, W. “A taxonomy of wireless micro-sensor network models”, *ACM SIGMOBILE Mobile Computing and Communications Review*, v. 6, n. 2, pp. 28–36, 2002.

[35] DENG, J., DONG, W., SOCHER, R., et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

[36] IWANA, B. K., UCHIDA, S. “An empirical survey of data augmentation for time series classification with neural networks”, *Plos one*, v. 16, n. 7, pp. e0254841, 2021.

[37] LASKARIDIS, S., VENIERIS, S. I., ALMEIDA, M., et al. “SPINN: synergistic progressive inference of neural networks over device and cloud”. In: *Proceedings of the 26th annual international conference on mobile computing and networking*, pp. 1–15, 2020.

[38] RAZZAQUE, M. A., BLEAKLEY, C., DOBSON, S. “Compression in wireless sensor networks: A survey and comparative evaluation”, *ACM Transactions on Sensor Networks (TOSN)*, v. 10, n. 1, pp. 1–44, 2013.

[39] KINGMA, D. P., BA, J. “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2014.

[40] NESTEROV, Y. E. “A method for solving the convex programming problem with convergence rate O (1/kˆ 2)”. In: *Dokl. akad. nauk Sssr*, v. 269, pp. 543–547, 1983.

[41] DOZAT, T. “Incorporating nesterov momentum into adam”. In: *International Conference on Learning Representations (ICLR)*, 2016.

[42] ABADI, M., AGARWAL, A., BARHAM, P., et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems”. 2015. Disponível em: <`https://www.tensorflow.org/`>. Software available from tensorflow.org.

[43] GILBERT, M. S. “Neural Network Temporal Compression”. `https://github.com/MateusGilbert/nn_temp_compression`, 2023. Accessed in 07/21/2023.

[44] DE PINHO, V. M., DE CAMPOS, M. L. R., GARCIA, L. U., et al. "Vision-aided radio: User identity match in radio and video domains using machine learning", *IEEE Access*, v. 8, pp. 209619–209629, 2020.

[45] EVERINGHAM, M., VAN GOOL, L., WILLIAMS, C. K. I., et al. "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results". http://host.robots.ox.ac.uk/pascal/VOC/voc2012/.

[46] PASZKE, A., GROSS, S., MASSA, F., et al. "Pytorch: An imperative style, high-performance deep learning library", *Advances in neural information processing systems*, v. 32, 2019.

[47] GILBERT, M. S. "EE-DNN for Semantic Segmentation: BranDeepLabV3". `https://github.com/MateusGilbert/brandeeplabv3`, 2023. Accessed in 01/16/2024.

[48] BERMAN, M., TRIKI, A. R., BLASCHKO, M. B. "The lovász-softmax loss: A tractable surrogate for the optimization of the intersection-over-union measure in neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4413–4421, 2018.

[49] MAINTAINERS, T., CONTRIBUTORS. "TorchVision: PyTorch's Computer Vision library". `https://github.com/pytorch/vision`, 2016.

[50] MEILĂ, M. "Comparing clusterings—an information based distance", *Journal of multivariate analysis*, v. 98, n. 5, pp. 873–895, 2007.

[51] CRESWELL, A., WHITE, T., DUMOULIN, V., et al. "Generative adversarial networks: An overview", *IEEE signal processing magazine*, v. 35, n. 1, pp. 53–65, 2018.

[52] RONNEBERGER, O., FISCHER, P., BROX, T. "U-net: Convolutional networks for biomedical image segmentation". In: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pp. 234–241. Springer, 2015.

# Appendix A

# List of Publications

Papers produced during the preparation of this M.Sc. thesis:

- **Gilbert, M. S.**, de Campos, M. L. R., Campista, M. E. M. - "Autoencoders Assimétricos para Compressão de Dados IoT", in XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2023), Brasília, Brazil, May 2023.
- **Gilbert, M. S.**, Pacheco, R. G., Couto, R. S., de Campos, M. L. R., Campista, M. E. M. - "Unlocking Early-Exiting Semantic Segmentation with Branched Networks", in 15th IEEE Latin-American Conference on Communications (LATINCOM 2023) Ciudad de Panamá, Panamá, November 2023.
- **Gilbert, M. S.**, de Campos, M. L. R., Campista, M. E. M. - "Asymmetric Autoencoders: an NN alternative for resource-constrained devices in IoT networks", in Ad Hoc Networks, April 2024.