POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

DEEP AND WIDE TINY MACHINE LEARNING

Doctoral Dissertation of:
Simone Disabato

Supervisor:
Prof. Manuel Roveri

Tutor:
Prof. Nicola Gatti

2021 – Cycle XXXIV

Abstract

In the last decades and, in particular, in the last few years, Deep Learning (DL) solutions emerged as state of the art in several domains, e.g., image classification, object detection, speech translation and command identification, medical diagnoses, natural language processing, artificial players in games, and many others.

In the same period, following the massive spread of pervasive technologies such as Internet of Things (IoT) units, embedded systems, or Micro-Controller Units (MCUs) in various application scenarios (e.g., automotive, medical devices, and smart cities, to name a few), the need for intelligent processing mechanisms as close as possible to data generation emerged as well. The traditional paradigm of having a pervasive sensor (or pervasive network of sensors) that acquires data to be processed by a remote high-performance computer is overcome by real-time requirements and connectivity issues.

Nevertheless, the memory and computational requirements characterizing deep learning models and algorithms are much larger than the corresponding abilities in memory and computation of embedded systems or IoT units, significantly limiting their application. The related literature in this field is highly fragmented, with several works aiming to reduce the complexity of deep learning solutions. However, only a few aim to deploy such DL algorithms on IoT units or even on MCUs. All these works fall under the umbrella of a novel research area, namely Tiny Machine Learning (TML), whose goal is to design machine and deep learning models and algorithms able to take into account the constraints on memory, computation, and also energy the embedded systems, the IoT, and the micro-controller units impose.

This work aims to introduce a methodology as well as algorithms and solutions to close the gap between the complexity of Deep Learning solutions and the capabilities of embedded, IoT, or micro-controller units.

Achieving this goal required operating at different levels. First, the methodology aims at proposing inference-based Deep Tiny Machine Learning solutions, i.e., DL algorithms that can run on tiny devices after their training has been carried out elsewhere. Second, the first approaches to on-device Deep Tiny Machine Learning training are proposed. Fi-

nally, the methodology encompasses Wide Deep TML solutions that distribute the DL processing on a network of embedded systems, IoT, and MCUs.

The methodology has been validated on available benchmarks and datasets to prove its effectiveness. Moreover, in a "from the laboratory to the wild" approach, the methodology has been validated in two different real-world scenarios, i.e., the detection of bird calls within audio waveforms in remote environments and the characterization and prediction of solar activity from solar magnetograms. Finally, a deep-learning-as-a-service approach to support privacy-preserving deep learning solutions (i.e., able to operate on encrypted data) has been proposed to deal with the need to acquire and process sensitive data on the Cloud.

Italian Summary

NEGLI ultimi decenni e in particolare negli ultimi anni, le soluzioni di *Deep Learning* sono velocemente diventate lo stato dell'arte in diversi scenari applicativi "intelligenti". Gli esempi più noti sono: la classificazione, il rilevamento e l'identificazione di oggetti nelle immagini; la classificazione di video e la creazione automatica di sottotitoli o descrizioni; la traduzione di discorsi; il riconoscimento di comandi vocali; le diagnosi mediche; l'analisi del linguaggio scritto; le intelligenze artificiali nei giochi; i sistemi di navigazione automatica delle automobili o dei droni; e i sistemi di raccomandazione ad esempio di film o prodotti in un mercato.

Nello stesso periodo, anche le tecnologie pervasive hanno vissuto una rapida espansione in vari scenari applicativi, come nei dispositivi medici; nelle automobili (ad esempio nella gestione degli airbag, nel mantenimento di una velocità di crociera, nel controllo di trazione e del sistema frenante); e nelle cosiddette *Smart Cities* (ovvero l'utilizzo di sistemi pervasivi in vari ambiti urbani, come la gestione dell'illuminazione pubblica, del trasporto pubblico o il monitoraggio ambientale). Esempi di dispositivi pervasivi sono i *sistemi embedded*, l'*Internet of Things* (IoT) e i *micro-controllori*, di seguito indicati per brevità come *unità IoT*. La necessità di spostare gli algoritmi intelligenti (ad esempio, per il riconoscimento di guasti o di cambiamenti nell'ambiente) il più vicino possibile al punto in cui i dati vengono generati è l'immediata conseguenza della diffusione pervasiva delle unità IoT.

Il paradigma tradizionale dove un sensore pervasivo acquisisce dati e li inoltra verso un server remoto (ad esempio, sul *Cloud*) –dove viene svolta tutta la computazione intelligente– in attesa di una risposta (ad esempio un comando per gli attuatori) non è più realistica e attuabile per tre motivi. In primo luogo, la connessione verso i server remoti deve essere stabile e ad alta velocità affinché l'intera soluzione sia attiva in maniera continuativa. In secondo luogo, delegare la computazione intelligente a un server remoto non è applicabile in tutti quegli scenari applicativi critici dove sono presenti requisiti stretti in termini di latenza tra l'acquisizione del dato e l'attuazione della corrispondente decisione. Infine, è preferibile non ricorrere a un server remoto laddove i dati che vengono acqui-

siti e inviati sono sensibili (ad esempio, quando vengono analizzate diagnosi mediche o immagini di persone in un sistema di video-sorveglianza).

Il problema principale nell'eseguire algoritmi intelligenti (e.g., basati su *Deep Learning*) su unità IoT è la complessità di quest'ultimi. Infatti, i requisiti in termini di memoria, computazione ed energia dei modelli di *Deep Learning* sono quasi sempre in contrasto con le corrispondenti capacità in termini di memoria, computazione ed energia disponibili nelle unità IoT. Per avere un'idea, i modelli convolutivi usati nel dominio delle immagini hanno decine di milioni di parametri (i modelli di *ResNet* hanno da 11 a 60 milioni di parametri, mentre quelli *Inception* da 24 a 43), mentre i modelli usati per modellare il linguaggio, come *BERT*, richiedono centinaia di milioni o miliardi di parametri. Siccome ogni parametro viene solitamente rappresentato con un tipo di dato a 32 bit, si osserva subito come la memoria richiesta soltanto per i parametri dei modelli scala facilmente da decine di mega-byte ai giga-byte. In termini di computazione richiesta, invece, il numero di operazioni richieste per classificare una singola immagine da modelli come la *ResNet* o l'*Inception* varia dai 5 agli 11 milioni. L'altra faccia della medaglia sono le capacità delle unità IoT. Ad esempio, il micro-controllore *STM32H743ZI* ha a disposizione 1024 kilo-byte di memoria e un processore Cortex M7 a 480 MHz, mentre gli altri micro-controllori hanno a disposizione dai 96 ai 512 kilo-byte di memoria e processori a frequenze minori.

In letteratura, questo problema è affrontato in maniera molto frammentata, con numerosi lavori che si pongono l'obiettivo di ridurre i requisiti di memoria o computazione delle soluzioni *Deep Learning*, ma con pochi lavori che hanno una visione d'insieme definendo quindi modelli di *Deep Learning* pensati per essere eseguiti su unità IoT.

In particolare, si possono individuare tre principali aree di ricerca. La prima si concentra nello sviluppo di soluzioni hardware dedicate. Le piattaforme hardware risultanti sono caratterizzate dalle migliori prestazioni in termini di latenza (tempo di esecuzione dell'algoritmo per cui sono state pensate), di consumi energetici e di potenza richiesta. Tuttavia il processo di sviluppo risulta particolarmente complesso e le soluzioni sviluppate sono caratterizzate da una minore flessibilità. La seconda area di ricerca introduce varie tecniche di approssimazione per ridurre la complessità in termini di memoria o di computazione dei modelli di *Deep Learning*. Esempi di tali tecniche sono: la riduzione della precisione nella rappresentazione dei parametri (da un tipo di dato a 32 bit verso rappresentazioni a 16, 8 e perfino 2 o 1 bit); l'introduzione di tecniche di *pruning* (letteralmente potare in inglese) sui parametri o su alcuni *task* dei modelli di *Deep Learning* stessi; e l'introduzione di uscite intermedie che possono essere prese quando il modello di *Deep Learning* acquisisce abbastanza confidenza sulla decisione finale, saltando di conseguenza tutta la computazione rimanente. Infine, la terza e ultima direzione di ricerca divide i modelli di *Deep Learning* in *task* semplici e compatibili con le unità IoT e studia il modo migliore per distribuire questi task su un insieme di unità IoT connesse e potenzialmente eterogenee.

Recentemente, è emersa una nuova area di ricerca, chiamata *Tiny Machine Learning* (TML), con l'obiettivo di sviluppare soluzioni di (*Machine* e) *Deep Learning* tenendo in considerazione i vincoli tecnologici dell'unità IoT su cui si pensa di eseguirle. L'impronta di memoria dei modelli TML generati deve essere quindi nell'ordine di grandezza di pochi kilo-byte, mentre il consumo energetico nell'ordine dei micro- o milli-Watt.

La maggior parte delle soluzioni TML disponibili in letteratura (come anche parte delle soluzioni presentate in questo lavoro) si focalizza sullo sviluppo di soluzioni a supporto dell'*inferenza* dei modelli di *Deep Learning*, ovvero la gestione di un singolo dato

in ingresso, come la classificazione di una immagine o la traduzione di una porzione di testo. Un'ulteriore direzione di ricerca, pressoché inesplorata in letteratura, ambisce a ideare soluzioni che permettono quello che in inglese viene definito come *on-device learning*, ovvero introdurre la possibilità di apprendimento per i modelli TML direttamente sull'unità IoT su cui vengono eseguiti. Il motivo di questa mancanza risiede principalmente nella particolare complessità delle tecniche di apprendimento rispetto alla semplice inferenza. Tuttavia, la capacità di apprendimento e quindi adattamento dei modelli TML direttamente sulle unità IoT è cruciale. L'ambiente in cui questi modelli operano è infatti tipicamente non stazionario, con effetti potenzialmente catastrofici sulle prestazioni dei modelli TML che assumono un ambiente immutabile nel tempo (esempi di cambiamenti che si riflettono nei dati acquisiti sono dovuti a guasti nei sensori di acquisizione, alla stagionalità o a effetti di invecchiamento).

L'obiettivo di questo lavoro è la definizione di una metodologia per lo sviluppo di soluzioni di *Deep and Wide Tiny Machine Learning*, dove l'aggettivo *deep* (profondo in inglese) suggerisce l'utilizzo dei modelli di *Deep Learning*, mentre il termine *wide* (largo, ampio in inglese) la possibilità di definire *task* da distribuire su più unità IoT potenzialmente eterogenee. In aggiunta, questo lavoro definisce una prima soluzione al problema dell'*on-device learning*.

La metodologia è stata validata sui *dataset* e sui *benchmark* disponibili per dimostrarne l'efficacia. Inoltre, secondo un approccio *from the lab to the wild* (dal laboratorio al selvaggio), alcune delle tecniche proposte sono state applicate in due scenari applicativi reali: il riconoscimento di vocalizzi di uccelli in aree remote (dove la connettività è assente o limitata) attraverso l'analisi di audio; e la caratterizzazione (e in futuro predizione) delle attività solari –altamente non stazionarie– attraverso l'analisi dei magnetogrammi solari acquisiti dalla Terra. Infine, in questo lavoro viene presentato un approccio *deep-learning-as-a-service* in cui l'esecuzione dei modelli di *Deep Learning* viene eseguita su dati criptati, per gestire tutti i casi non coperti dalla metodologia presentata in cui è necessario utilizzare servizi sul *Cloud*, ma al tempo stesso garantire la privacy dei dati che vengono elaborati.

Contents

List of Figures

List of Tables

List of Algorithms

Part Ⅰ

Introduction

CHAPTER 1

Introduction to Tiny Machine Learning

1.1 Closing the Gap Between Deep Learning and IoT Units, Embedded Systems, and Micro-controllers

Machine Learning (ML) and Deep Learning (DL) techniques have widely spread across the most diverse areas in the last decade, achieving state of the art in several fields. Convolutional Neural Networks (CNNs) models, such as the ResNet (He et al., 2016) or the Inception (Szegedy et al., 2015; 2017), provide a classification of input images. Extensions of such architectures, such as the Yolo (Redmon et al., 2016; Redmon and Farhadi, 2018) or the EfficientDet (Tan et al., 2020), also identify the position of detected objects within images. Recurrent DL architectures achieve the highest classification capabilities in video classification (Tran et al., 2019), but also in speech recognition, translation (Baevski et al., 2019; Zhang et al., 2018b), and language modelling (Devlin et al., 2019; Shoeybi et al., 2019). Other DL techniques have been also successfully applied to different fields, such as anomaly detection (Tack et al., 2020), recommender systems (Muller et al., 2018; Zheng et al., 2016), reinforcement learning (Li, 2018; Metelli et al., 2019), autonomous driving (Chen et al., 2015) and drones navigation (Gandhi et al., 2017).

A few characteristics can be found to be commonly shared among all of these DL techniques. First of all, a high number of parameters. For example, the ResNet CNN has 11 to 60 million parameters, the Inception 24 to 43 million, whereas the language modeling techniques, e.g., BERT (Devlin et al., 2019), have hundreds of millions or even billions of parameters. By considering a 32-bit (floating-point) data type, the memory required by these parameters easily scales from dozens of megabytes to several gigabytes.

Furthermore, performing the training of these models can require days, weeks, or even more on high-performance computers, often equipped with clusters of GPUs, as deeply analyzed in (Chen et al., 2018a).[1] Finally, performing inference of such DL models, i.e., the processing of one –unseen– input by a trained model (e.g., the classification of a new image or the sentence identification in an audio sample), is significantly simpler than the training. However, it still requires millions or billions of floating-point operations per single input. For instance, the Inception and the Resnet need 5 to 11 billion multiplications to classify a single input image.

Among all the above-mentioned ML and DL techniques' possible applications, a challenging and breakthrough technology with enormous room for improvement is the so-called *intelligence for pervasive units*, such as IoT units or embedded systems. Such devices are nowadays part of our everyday life in a wide range of application scenarios (e.g., smart cities, automotive, or medical devices) and ask to move the processing (and in particular the intelligent processing) as close as possible to where data are generated. Indeed, machine and deep learning solutions processing these data directly on the pervasive devices are crucial to support real-time applications, prolong the system lifetime, and increase the Quality-of-Service (Alippi et al., 2018; Sanchez-Iborra and Skarmeta, 2020; Tang et al., 2017a). The downside is that IoT units and embedded systems have strict constraints on memory, computation, and power consumption. The order of magnitude is significantly lower than that required by DL solutions, being Kilobytes to (a few) Megabytes in terms of memory (Disabato et al., 2021b) and milli-Watts in terms of power consumption (Kim et al., 2018; Martinez et al., 2015). Such constraints are even harsher and more severe on Micro-Controllers Units (MCUs). Their memory is indeed of a few Kilo-Bytes, with expected power consumptions of micro- to milli-Watts (Banbury et al., 2020; Lin et al., 2020a). A few examples follow. The high-performance STM32H743ZI MCU (of STMicroelectronics), equipped with a 480-MHz Cortex M7 processor, has 1024KB of RAM (divided into five blocks of different speed). The majority of other ST MCUs have 96 to 512KB of RAM. The MCUs of Texas Instrument (e.g., the TMS320F280025C or the Cortex-M4–based TM4C1294NCPDT) are usually equipped with 128 or 256KB of RAM.

The goal of this thesis –and of any work addressing this problem– is to design intelligent mechanisms based on machine or deep learning techniques whose requirements in terms of memory footprint, computational load, and energy are compatible with the technological constraints on memory, computation, and energy introduced by the IoT units and the embedded systems they are designed for. Chapter 3 widely analyses the related literature, highlighting that it is highly fragmented and with very few solutions encompassing all the aspects of this problem (Alippi et al., 2018; Teerapittayanon et al., 2017). However, there are three major research directions.

At first, several works design dedicated hardware solutions for machine and deep learning models (see Section 3.1). Such solutions provide significant gains in terms of power consumption and performances with respect to general hardware at the expense of a complex design phase of the design phase and a reduced flexibility (Cavigelli and Benini, 2016; Zhang et al., 2015a; Dundar et al., 2016).

A second major research direction is that of approximated solutions (see Section 3.2)

[1] As an example, (Richards et al., 2021) estimates about 4.5 years in terms of GPU hours to properly train their deep learning model, whereas (Badre et al., 2018) estimates about 16 (GPU-)weeks to train the Inception-V3 (Szegedy et al., 2016) Convolutional Neural Network.

with several different approaches to reduce the complexity of deep learning models. In almost all those solutions, the IoT or microcontroller units are not a target of the introduced approximations. Task dropping (e.g., pruning of deep learning models' layers or parameters) (Denton et al., 2014; Han et al., 2016; He et al., 2017; Li et al., 2018; Lin et al., 2018; Rigamonti et al., 2013) and quantization techniques on parameters (Bulat and Tzimiropoulos, 2021; Cai et al., 2017; Gupta et al., 2015; Rastegari et al., 2016) are able to (drastically) reduce the memory footprint of machine and deep learning models at the expenses of a drop in the metric the algorithm is evaluated on. Similarly, the introduction of early-exit paths within such algorithms allows reducing their mean computational complexity (Bolukbasi et al., 2017; Disabato and Roveri, 2018; Kaya et al., 2019; Yang et al., 2020; Zhou et al., 2019; 2020), with an impact strictly dependent on how frequently such early-exit paths are taken (thus skipping the remaining computation).

Finally, the third research direction is that of distributed computation (see Section 3.5), with solutions derived from the offloading solutions aiming at finding the optimal distribution of the processing pipeline of deep learning models across a set of heterogeneous IoT units, MCUs, and, if any, the Cloud (Bhardwaj et al., 2019; Chen et al., 2019a; Disabato et al., 2021b; Hu and Krishnamachari, 2020; Tao and Li, 2018; Teerapittayanon et al., 2017; Zhao et al., 2018).

Recently, Tiny Machine Learning (TML) (Banbury et al., 2020; Cai et al., 2020a; Disabato and Roveri, 2021; Fedorov et al., 2019; 2020; Gopinath et al., 2019; Kumar et al., 2017; Lin et al., 2020a; Rusci et al., 2020; Venzke et al., 2020) emerged as a novel and promising further research direction aiming at designing machine and deep learning solutions able to be executed on IoT units or even on micro-controller units (namely, tiny devices), i.e., with a memory footprint in the order of kilobytes and power consumption in the order of milli- to micro-Watts. To fill the gap between the memory, computational, and energy demands of machine and deep learning models with the corresponding requirements of tiny devices, all the techniques coming from the literature of approximated deep learning mentioned above are brought into play.

Furthermore, the TML solutions mainly enable the inference of DL or ML algorithms. There are indeed very few works in the field of Tiny Machine Learning proposing on-device learning, i.e., the training of machine and deep learning solutions directly on the tiny devices (Cai et al., 2020a; Disabato and Roveri, 2020; 2021). The ability to learn TML models directly on tiny devices is crucial to improve the TML algorithm over time by exploiting fresh information coming from the field, and to deal with concept drift, i.e., variations in the statistical behavior of the data generating process, a quite common situation in real-world applications (e.g., due to seasonality or periodicity effects, faults affecting sensors or actuators, changes in the user's behavior, or aging consequences). Failing to adapt TML models to concept drift results in a (possibly dramatic) decrease of the TML accuracy over time (Ditzler et al., 2015; Disabato et al., 2021b).

In this challenging scenario, this thesis proposes a methodology to design and deploy Deep and Wide Tiny Machine Learning (TML) solutions that are able to take into account the constraints on memory, computation, and energy of tiny devices (either IoT units, MCUs, or Embedded Systems). More in detail, the methodology addresses the problem from two different perspectives, each of which is deepened in a specific part of this manuscript:

- Part II details how to support the inference of Deep Learning Models (DLMs) on

Tiny devices, i.e., the design of Deep Tiny Machine Learning solutions by means of approximation (Chapter 5). More in details, the approximation encompasses either the memory footprint of the DL algorithm (Section 5.1) or its mean computational complexity (Section 5.2) by introducing one or more early-exits (Gate-Classifiers). In addition to stationary Deep TML models, Section 5.3 introduces adaptation techniques to allow these Tiny Deep Learning Model to react to concept drift. Similarly, Chapter 6 presents the proposed solution for on-device learning.

- Part III focuses on the Wide (Deep) Tiny Machine Learning algorithms, i.e., it addresses (in Chapter 7) the problem mentioned above by splitting the DLMs into (non-approximated) sub-tasks then distributed among possibly heterogeneous tiny devices. In this way, wider DLMs can be designed at the expense of taking into account the introduced communication issues and delays. The term *Wide* here should not be confused with the so-called Wide and Deep Learning models (Burel et al., 2017; Cheng et al., 2016; Zheng et al., 2017) that combine a wide linear model and a Deep Learning one. Moreover, Chapter 8 suggests a solution to run a part of DLM computation on the Cloud in a confidential manner, i.e., in such a way the Cloud can produce the expected output without knowing what it is processing about.

The remaining of the thesis is organized as follows:

- Part I is the opening part of the thesis. In addition to this introductory Section, Section 1.2 details the scientific papers presented in this manuscript as well as the corresponding Chapters or Sections presenting them, whereas Chapter 2 formalizes the complexity of Machine and Deep Learning Models. Then, Chapter 3 widely inspects the related literature, whereas Chapter 4 formalizes the addressed problem of designing Deep and Wide Tiny Machine Learning solutions.

- Part IV summarizes the experimental results of this work, proving the effectiveness of the proposed work.

- Part V tailors the theory of the previous parts to two real application scenarios, showing the feasibility and effectiveness of the proposed methodology. In particular, Section 11.1 presents two Deep Tiny Machine Learning solutions to detect bird calls or songs in 10s length audios that are able to run in a remote area for more than a week. Section 11.2 deals with the highly nonstationary processes at the Sun, proposing solutions to model the solar activity within acquired magnetograms.

1.2 Paper Contributions

This section summarizes the research contributions and points them to the corresponding portion of the manuscript where they are presented.

Part II–Chapter 5 encompasses the following works (Alippi et al., 2018; Disabato and Roveri, 2018; 2019):

(i) Cesare Alippi, Simone Disabato, and Manuel Roveri. Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 212–223, Porto, apr 2018. IEEE

Abstract

Execution of deep learning solutions is mostly restricted to high performing computing platforms, e.g., those endowed with GPUs or FPGAs, due to the high demand on computation and memory such solutions require. Despite the fact that dedicated hardware is nowadays subject of research and effective solutions exist, we envision a future where deep learning solutions -here Convolutional Neural Networks (CNNs)- are mostly executed by low-cost off-the shelf embedded platforms already available in the market.

This paper moves in this direction and aims at filling the gap between CNNs and embedded systems by introducing a methodology for the design and porting of CNNs to limited in resources embedded systems. In order to achieve this goal we employ approximate computing techniques to reduce the computational load and memory occupation of the deep learning architecture by compromising accuracy with memory and computation.

The proposed methodology has been validated on two well-know CNNs, i.e., AlexNet and VGG-16, applied to an image-recognition application and ported to two relevant off-the-shelf embedded platforms.

(ii) Simone Disabato and Manuel Roveri. Reducing the computation load of convolutional neural networks through gate classification. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

Abstract

Reducing the computational load of Convolutional Neural Networks (CNNs) is of utmost importance to allow their execution in computing systems characterized by constraints on computation and energy (e.g., embedded and cyber-physical systems and Internet-of-Things). To address this problem, which has been rarely addressed in the related literature, this paper introduces the Gate-Classification CNNs. The core of this novel family of CNNs is the presence of Gate-Classification layers that allow to incrementally process the input image through the CNN layers and take a decision as soon as "enough confidence" about the classification is gained, hence not requiring the processing of the whole CNN when not needed. The Gate-Classification CNNs rely on the ability of CNNs to process features characterized by increasing complexity and meaning and, in particular, the Gate-Classification layers allow to select the path within the CNN according to the information content provided by the input image and the processed features. A wide experimental campaign on public-available datasets supports the effectiveness of the proposed solution.

(iii) Simone Disabato and Manuel Roveri. Learning convolutional neural networks in

presence of concept drift. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

Abstract

Designing adaptive machine learning systems able to operate in nonstationary conditions, also called concept drift, is a novel and promising research area. Convolutional Neural Networks (CNNs) have not been considered a viable solution for such adaptive systems due to the high computational load and the high number of images they require for the training. This paper introduces an adaptive mechanism for learning CNNs able to operate in presence of concept drift. Such an adaptive mechanism follows an "active approach", where the adaptation is triggered by the detection of a concept drift, and relies on the "transfer learning" paradigm to transfer (part of the) knowledge from the CNN operating before the concept drift to the one operating after. The effectiveness of the proposed solution has been evaluated on two types of CNNs and two real-world image benchmarks.

Part III–Chapter 6 encompasses the following works (Disabato and Roveri, 2020; 2021):

(iv) Simone Disabato and Manuel Roveri. Incremental on-device tiny machine learning. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, pages 7–13, 2020.

Abstract

Tiny Machine Learning (TML) is a novel research area aiming at designing and developing Machine Learning (ML) techniques meant to be executed on Embedded Systems and Internet-of-Things (IoT) units. Such techniques, which take into account the constraints on computation, memory, and energy characterizing the hardware platform they operate on, exploit approximation and pruning mechanisms to reduce the computational load and the memory demand of Machine and Deep Learning (DL) algorithms.

Despite the advancement of the research, TML solutions present in the literature assume that Embedded Systems and IoT units support only the inference *of ML and DL algorithms, whereas their* training *is confined to more-powerful computing units (due to larger computational load and memory demand). This also prevents such pervasive devices from being able to learn in an incremental way directly from the field to improve the accuracy over time or to adapt to new working conditions.*

The aim of this paper is to address such an open challenge by introducing an incremental algorithm based on transfer learning and k-nearest neighbor to support the on-device learning (and not only the inference) of ML and DL solutions on embedded systems and IoT units. Moreover, the proposed solution is general and can be applied to different application scenarios. Experimental results on image/audio benchmarks and two off-the-shelf hardware platforms show the feasibility and effectiveness of the proposed solution.

(v) Simone Disabato and Manuel Roveri. Tiny machine learning for concept drift. *arXiv preprint arXiv:2107.14759*, 2021.

Abstract

Tiny Machine Learning (TML) is a new research area whose goal is to design machine and deep learning techniques able to operate in Embedded Systems and IoT units, hence satisfying the severe technological constraints on memory, computation, and energy characterizing these pervasive devices. Interestingly, the related literature mainly focused on reducing the

computational and memory demand of the inference phase of machine and deep learning models. At the same time, the training is typically assumed to be carried out in Cloud or edge computing systems (due to the larger memory and computational requirements). This assumption results in TML solutions that might become obsolete when the process generating the data is affected by concept drift (e.g., due to periodicity or seasonality effect, faults or malfunctioning affecting sensors or actuators, or changes in the users' behavior), a common situation in real-world application scenarios. For the first time in the literature, this paper introduces a Tiny Machine Learning for Concept Drift (TML-CD) solution based on deep learning feature extractors and a k-nearest neighbors classifier integrating a hybrid adaptation module able to deal with concept drift affecting the data-generating process. This adaptation module continuously updates (in a passive way) the knowledge base of TML-CD and, at the same time, employs a Change Detection Test to inspect for changes (in an active way) to quickly adapt to concept drift by removing the obsolete knowledge. Experimental results on both image and audio benchmarks show the effectiveness of the proposed solution, whilst the porting of TML-CD on three off-the-shelf micro-controller units shows the feasibility of what is proposed in real-world pervasive systems.

Part III–Chapter 7 presents the following paper (Disabato et al., 2021b):

(vi) Simone Disabato, Manuel Roveri, and Cesare Alippi. Distributed deep convolutional neural networks for the internet-of-things. *IEEE Transactions on Computers*, 2021b.

Abstract

Severe constraints on memory and computation characterizing the Internet-of-Things (IoT) units may prevent the execution of Deep Learning (DL)-based solutions, which typically demand large memory and high processing load. In order to support a real-time execution of the considered DL model at the IoT unit level, DL solutions must be designed having in mind constraints on memory and processing capability exposed by the chosen IoT technology. In this paper, we introduce a design methodology aiming at allocating the execution of Convolutional Neural Networks (CNNs) on a distributed IoT application. Such a methodology is formalized as an optimization problem where the latency between the data-gathering phase and the subsequent decision-making one is minimized, within the given constraints on memory and processing load at the units level. The methodology supports multiple sources of data as well as multiple CNNs in execution on the same IoT system allowing the design of CNN-based applications demanding autonomy, low decision-latency, and high Quality-of-Service.

Part III–Chapter 8 encompasses the paper on privacy preserving (Disabato et al., 2020):

(vii) Simone Disabato, Alessandro Falcetta, Alessio Mongelluzzo, and Manuel Roveri. A privacy-preserving distributed architecture for deep-learning-as-a-service. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

Abstract

Deep-learning-as-a-service is a novel and promising computing paradigm aiming at providing machine/deep learning solutions and mechanisms through Cloud-based computing infrastructures. Thanks to its ability to remotely execute and train deep learning models (that typically require high computational loads and memory occupation), such an approach

guarantees high performance, scalability, and availability. Unfortunately, such an approach requires to send information to be processed (e.g., signals, images, positions, sounds, videos) to the Cloud, hence having potentially catastrophic-impacts on the privacy of users. This paper introduces a novel distributed architecture for deep-learning-as-a-service that is able to preserve the user sensitive data while providing Cloud-based machine and deep learning services. The proposed architecture, which relies on Homomorphic Encryption that is able to perform operations on encrypted data, has been tailored for Convolutional Neural Networks (CNNs) in the domain of image analysis and implemented through a client-server REST-based approach. Experimental results show the effectiveness of the proposed architecture.

Finally, Part V–Chapter 11 presents the paper on birdsong detection (Disabato et al., 2021a) and that on solar magnetograms (Valdés et al., 2020):

(viii) Simone Disabato, Giuseppe Canonaco, Paul G Flikkema, Manuel Roveri, and Cesare Alippi. Birdsong detection at the edge with deep learning. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 9–16. IEEE, 2021a.

Abstract

Understanding the distribution of bird species and populations and learning how birds behave and communicate are of great importance in wildlife biology, animal ecology, conservation of ecosystems, and assessing the effects of climate change and urbanization. The temporal and spatial limitations of human observation have motivated significant efforts to develop technology for bird song and vocalization detection and classification. While solutions based on signal processing and machine learning are extant, they are limited in various combinations of speed, computational complexity, and memory use, as well as in detection/-classification capability in real-world conditions. This paper introduces ToucaNet, a deep neural network for birdsong detection based on transfer-learning, a deep learning mechanism allowing us to exploit knowledge acquired on various tasks: this enables us to speed up training and shows improved detection accuracy. ToucaNet provides birdsong detection accuracy in line with the best solutions in the literature but with much less computational complexity and memory demand. We also introduce BarbNet, an approximated version of ToucaNet tailored for Internet-of-Things (IoT) units. We show the proposed solution's effectiveness and efficiency in terms of detection accuracy and the implementation feasibility in real-world IoT devices, with specific results for the STM32 Nucleo H7 board, which is based on an ARM Cortex-M7 processor. To our best knowledge, this is the first birdsong detection algorithm designed to take into account constraints on memory, computational speed, and power usage of embedded devices. Thus, this work points the way to cost-effective IoT technology for at-scale intelligent birdsong data collection and analysis in the field.

(ix) Julio J Valdés, Ljubomir Nikolić, Simone Disabato, and Manuel Roveri. A computational intelligence characterization of solar magnetograms. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020 .

Abstract

Space Weather (SW) poses a hazard to modern society. SW phenomena depend on the Sun's magnetic field, whose understanding and forecasting is an important research subject. To achieve this goal, in this paper Global Oscillation Network Group (GONG) solar magnetograms 2006-2019 are investigated with different approaches provided by unsupervised and

supervised Computational Intelligence techniques. Such techniques were successful at providing insights into the behavior and evolution of the photospheric magnetic field, revealing patterns of activity and their relation with the different phases of the solar cycle. On the one hand, representative prototypes of synoptic maps were found, capturing the variations in homogeneity, intensity and variability of magnetic activity. On the other hand, Convolutional Neural Networks (CNNs) combined with transfer learning and dimensionality reduction techniques were helpful in providing classification models that accurately predict classes associated with the main stages of the cycle. Such models provide results in good correspondence with the natural classes found in feature spaces and have classification errors concentrated mostly at transition periods of the solar cycles.

CHAPTER 2

Background: Complexity of Deep Learning Models

This section aims at presenting and detailing the complexity of the deep learning models, by extending the formalizations proposed in (Alippi et al., 2018; He and Sun, 2015).

Let $\mathcal{D}_\vartheta$ be a deep learning model with parameters ϑ (the formal definition can be found in Section 4.2). The **memory complexity** (or memory footprint) $m_\mathcal{D}$ is then defined as the quantity of memory required to store all the parameters, i.e.,

$$m_\mathcal{D} = |\vartheta| \cdot m_\vartheta, \tag{2.1}$$

where $|\vartheta|$ is the cardinality of ϑ (i.e., the number of parameters of $\mathcal{D}$), and m_ϑ the memory footprint of each parameter.[2] Typically, $m_\vartheta = 32$ bits with the corresponding single-precision floating-point representation (Gupta et al., 2015). However, as shown in Section 3.2.2, m_ϑ can be reduced by relying on half-precision floating-point (16 bits), fixed-point, 8-bit integer, and even binary (1-bit) representations for the parameters. Section 2.1 details the number of parameters characterizing several deep learning layers.

It is worth noting that, when deploying a deep learning model to an IoT device or an MCU, the above-proposed definition of the memory complexity $m_\mathcal{D}$ is an underestimation of the effective memory required to run inference. A more detailed definition, not considered in this work, is the following:

$$\hat{m}_\mathcal{D} = |\vartheta| \cdot m_\vartheta + m_{in} + m_{aux}, \tag{2.2}$$

[2] In the following the number of parameters of a given layer ℓ within $\mathcal{D}$ is referred to as p_ℓ.

where the parameters ϑ footprint is summed to that of the input m_{in} and the footprint m_{aux} of all the auxiliary data structures needed by $\mathcal{D}$, including the activations (i.e., the transformations of the input at the end of each layer of $\mathcal{D}$). This formalization is less general than that in Equation (2.1) since the term m_{aux} strictly depends on the specific implementation of the model $\mathcal{D}$. However, it better depicts the footprint of the $\mathcal{D}$ inference, allowing an easy match to the available RAM of an IoT unit or an MCU. To run the $\mathcal{D}$ inference on such a device is indeed enough that its RAM capacity is at least $\hat{m}_{\mathcal{D}}$.

The **time complexity** (or computational complexity or computational cost) is defined as in (Alippi et al., 2018; He and Sun, 2015) as the number of *multiply-add* operations to carry out one inference of $\mathcal{D}$, i.e.,

$$c_{\mathcal{D}} = n_{mul} + n_{add}, \tag{2.3}$$

This definition in terms of number of operations is the most general and has the advantage of being independent from the platform where the deep learning model $\mathcal{D}$ is executed on, its operating system, its memory management system, and all the other technological aspects that influence the execution time. However, an estimation of the inference time $t_i^{(\mathcal{D})}$ required by model $\mathcal{D}$ can be obtained as follows:

$$t_i^{(\mathcal{D})} = n_{mul} \cdot t_{mul} + n_{add} \cdot t_{add} + c_{\mathcal{D}} \cdot t_{mem}, \tag{2.4}$$

where t_{mul} and t_{add} represent the time demanded to carry out a multiplication and an addition, respectively; whereas t_{mem} encompasses all the timings needed *in average* to move the operands (of both multiplications and additions) into the RAM and then into the arithmetic unit ($t_{mem} = 0$ when all the parameters are stored into RAM). It is crucial to point out that all the three terms t_{mul}, t_{add}, and t_{mem} depend on the architecture running the deep learning model and on the data type of the operands. Moreover, this formulation does not consider any optimization (e.g., caching mechanisms) or delays due to the operating system's scheduling. As a consequence, from now on, the time complexity is defined as in Equation (2.3). Section 2.2 details the time complexity of several deep learning layers.

Finally, the **energy complexity** $e_{\mathcal{D}}$ is defined as an extension of the time complexity, i.e.,

$$e_{\mathcal{D}} = n_{mul} \cdot e_{mul} + n_{add} \cdot e_{add}, \tag{2.5}$$

where e_{mul} and e_{add} represent the energy consumption required by a multiplication and an addition operation, respectively. Differently from the memory and the time complexity, this complexity term cannot be generalized to be implementation-independent. The terms e_{mul} and e_{add} are indeed relative to the architecture the model $\mathcal{D}$ is deployed on and on the parameters (and activations) representation. Section 2.3 details the energy consumption terms e_{mul} and e_{add} on target MCUs for different data types.

2.1 Characteristics and Number of Parameters in DL Layers

This section details the characteristics and the number of parameters for various kinds of Deep Learning layers.

Table 2.1: *The memory and time complexity of several deep learning models layers. Please refer to Section 2 for their formal definition. Unless otherwise specified the layers are supposed to operate on three-dimensional input $x_{in} \cdot y_{in} \cdot z_{in}$ and generate an output of size $x_o \cdot y_o \cdot z_o$.*

Layer ℓ	Layer Parameters	Memory Complexity p_ℓ	Time Complexity c_ℓ $(O(\cdot))$
Convolutional	Number of filters f_n Filter size $f_x \cdot f_y \cdot f_z$	$f_x \cdot f_y \cdot f_z \cdot f_n$	$f_x \cdot f_y \cdot z_{in} \cdot f_n \cdot x_o \cdot y_o$
Pooling	Pooling Sizes p_x, p_y	0	$p_x \cdot p_y \cdot x_o \cdot y_o \cdot z_o$
Batch Normalization	γ_c, β_c $(c = 0, \ldots, z_{in})$	$2 \cdot z_{in}$	$x_{in} \cdot y_{in} \cdot z_{in}$
Local Response Normalization	Number of channels n_c α, β	3	$n_c \cdot x_{in} \cdot y_{in} \cdot z_{in}$
Linear (input x_{in})	Number of outputs x_o	$x_{in} \cdot x_o$	$x_{in} \cdot x_o$

2.1.1 Convolutional Layer

The (three-dimensional) convolutional layer is typical of Convolutional Neural Networks (CNNs). Given a three-dimensional input $x_{in} \times y_{in} \times z_{in}$, it applies f_n convolutional filters (of size $f_x \cdot f_y \cdot f_z$) repeatedly on it, with $f_z = z_{in}$ in almost all the applications. In details, the convolution starts from the upper left portion of the input and then slides by a parameter called stride (s_x on rows, s_y on columns). The resulting output has size $x_o \times y_o \times z_o$, where:

$$x_o = \frac{x_{in} - \bar{f}_x + 2 \cdot pad_x}{s_x} + 1, \quad y_o = \frac{y_{in} - \bar{f}_y + 2 \cdot pad_y}{s_y} + 1, \quad z_o = f_n, \tag{2.6}$$

where pad_x and pad_y represent the (zero-)padding on both sides on rows and columns, respectively, and $\bar{f}_x = (f_x - 1) \cdot d_x + 1$ and $\bar{f}_y = (f_y - 1) \cdot d_y + 1$ represent the filter size when employing dilation d_x and d_y. Dilation is a convolutional parameter that defines the distance in terms of input values along the given dimension between two consecutive values of the filters (almost all the applications uses $d_x = d_y = 1$ that means the filter is applied on adjacent input values).

The number of parameters p_{conv} is

$$p_{conv} = f_x \cdot f_y \cdot f_z \cdot f_n, \tag{2.7}$$

that are the parameters of the filters themselves. It is noteworthy to point out that these results hold for three-dimensional convolutional layers but can be easily adapted to convolutional layers with two or more than three dimensions.

Implementing such a layer usually relies on a matrix multiplication between the (suitably modified) input and the matrix of filters. The availability of many routines to carry out the matrix multiplication is the reason behind this choice (Burrus and Parks, 1985; Chetlur et al., 2014; Sze et al., 2017). The input is converted into a matrix with $x_o \cdot y_o \cdot z_o$ rows and $f_x \cdot f_y \cdot f_z$ columns, where each row is an unrolled sequence of the input the convolutional filter is multiplied by. The matrix of filter has $f_x \cdot f_y \cdot f_z$ rows and f_n columns, where each column is an unrolled filter. Since the input patches covered by the convolutional filters are overlapping (unless the rare cases where the stride (at least) equals the filter size), the

just mentioned conversion into matrix multiplication results in a (non-negligible) memory footprint increment that should be considered when using this algorithm. Consequently, a balance between the gain in terms of computation speed and the memory occupation is required.

2.1.2 Maximum or Average Pooling Layer

The pooling layers are common solutions in deep learning architectures to reduce the dimensionality of the activations. Given a three-dimensional input $x_{in} \times y_{in} \times z_{in}$, they apply their pooling function (i.e., either maximum or average) on bidimensional windows of size $p_x \times p_y$. In detail, the pooling operates as convolutional layers starting from the upper left portion of the input, then sliding by the stride (namely s_x and s_y). However, it considers each channel independently (where channels refer to the third dimension of the input, i.e., z_{in}). The resulting output has size $x_o \times y_o \times z_o$, where:

$$x_o = \frac{x_{in} - p_x + 2 \cdot pad_x}{s_x} + 1, \quad y_o = \frac{y_{in} - p_y + 2 \cdot pad_y}{s_y} + 1, \quad z_o = z_{in}, \tag{2.8}$$

where pad_x and pad_y are the same of convolutional layers.

The pooling layers, being dimensionality reduction operators, do not introduce parameters, i.e.,

$$p_{pool} = 0. \tag{2.9}$$

As a final remark, there are less common pooling layers that apply the power of degree p to the inputs (where $p = \infty$ corresponds to the maximum function) or stochastic formulations in the definition of pooling regions (Graham, 2014).

2.1.3 Batch and Local Normalization Layer

The normalization layers are common in many state-of-the-art CNNs, with a few variants. The most common is *batch-normalization* (Ioffe and Szegedy, 2015) that normalizes each channel (the third dimension in a three-dimensional input) by subtracting the mean and by dividing by the variance of either the batch itself or the moving average values of the batches seen so far. In addition, the affine version of this transform adds to learnable parameters per each channel c, namely β_c and γ_c, defining the following normalization:

$$x_c = \frac{x_c - \mathcal{E}[x_c]}{\sqrt{Var[x_c] + \epsilon}} \cdot \gamma_c + \beta_c, \tag{2.10}$$

where ϵ is a numerical stability small constant value and x_c here refers to the c-th channel of the input (after and before the normalization).

The *local response normalization*, used for instance in the AlexNet (Krizhevsky et al., 2012), normalizes instead each channel according to the values of nearby channels as follows:

$$x_c = x_c \cdot \left(\kappa + \frac{\alpha}{n_c} \cdot \sum_{c'=\max\{0, c-n_c/2\}}^{\min\{z_{in}, c+n_c/2\}} x_{c'}^2 \right)^{-\beta}, \tag{2.11}$$

where n_c is the number of nearby channels the transform considers (if available), and α, β, and κ are the normalization parameters.

For each type of normalization, the output has the same size of the input, i.e., $x_o \times y_o \times z_o = x_{in} \times y_{in} \times z_{in}$. The number of parameters strictly depends on the type of normalization:

$$p_{batch\_norm} = 2 \cdot z_{in}, \qquad p_{local\_response\_norm} = 3, \tag{2.12}$$

that are negligible values w.r.t. other layers, as convolutional or fully-connected ones.

2.1.4 Linear or Fully-Connected Layer

The linear or fully connected layer is the most straightforward layer adopted from the first artificial neural networks. Given an input of size x_{in} and a number of outputs (usually referred to as hidden neurons) x_o, the output is a linear combination of all the inputs. The number of parameters is thus:

$$p_{fc} = x_{in} \cdot x_o. \tag{2.13}$$

Since the layer is a matrix multiplication between the parameters' matrix (of size $x_{in} \cdot x_o$) and the layer's input, all the implementations usually rely on matrix multiplication routines as in convolutional layers.

2.2 Number of Operations in Several DL Layers

2.2.1 Convolutional Layer

Given a three-dimensional input $x_{in} \times y_{in} \times z_{in}$, the (three-dimensional) convolutional output has size $x_o \times y_o \times z_o$ as defined in Eq. (2.6), computed with the following time complexity with f_n filters of size $f_x \cdot f_y \cdot z_{in}$:

$$\begin{aligned} t_{mul\_conv} &= f_x \cdot f_y \cdot z_{in} \cdot f_n \cdot x_o \cdot y_o, \\ t_{add\_conv} &= (f_x \cdot f_y \cdot z_{in} - 1) \cdot f_n \cdot x_o \cdot y_o, \end{aligned} \tag{2.14}$$

that is $O\left(f_x \cdot f_y \cdot z_{in} \cdot f_n \cdot x_o \cdot y_o\right)$ in the worst case (i.e., not considering matrix multiplication-based algorithm detailed in Section 2.1.1).

It is noteworthy to point out that the formalization above does not explicitly show the values of stride, padding, and dilation that are embedded into the output size values.

2.2.2 Maximum or Average Pooling Layer

Given a three-dimensional input $x_{in} \times y_{in} \times z_{in}$, a pooling function with a windows of size $p_x \times p_y$ produces an output of size $x_o \times y_o \times z_o$ as defined in Eq. (2.8), with the following time complexity:

$$\begin{aligned} t_{mul\_avg\_pool} &= p_x \cdot p_y \cdot x_o \cdot y_o \cdot z_o, \\ t_{add\_avg\_pool} &= (p_x \cdot p_y - 1) \cdot x_o \cdot y_o \cdot z_o, \end{aligned} \tag{2.15}$$

in the case of average pooling and

$$\begin{aligned} t_{mul\_max\_pool} &= 0, \\ t_{add\_max\_pool} &= p_x \cdot p_y \cdot x_o \cdot y_o \cdot z_o, \end{aligned} \tag{2.16}$$

in the case of maximum pooling, with the number of additions that counts all the logical comparisons to carry out the maximum function (the number of additions is 0).

The complexity of a pooling layer is then $O(p_x \cdot p_y \cdot x_o \cdot y_o \cdot z_o)$.

2.2.3 Batch and Local Normalization Layer

The batch normalization is a linear transformation that, given a three-dimensional input $x_{in} \times y_{in} \times z_{in}$, has the following time complexity:

$$\begin{aligned} t_{mul\_batch\_norm} &\approx 3 \cdot x_{in} \cdot y_{in} \cdot z_{in}, \\ t_{add\_max\_pool} &\approx 3 \cdot x_{in} \cdot y_{in} \cdot z_{in}, \end{aligned} \tag{2.17}$$

that corresponds to $O(x_{in} \cdot y_{in} \cdot z_{in})$.

The *local response normalization*, with n_c channels to compare with, is slightly more complex having a time complexity:

$$\begin{aligned} t_{mul\_local\_response\_norm} &= (n_c + 3) \cdot x_{in} \cdot y_{in} \cdot z_{in}, \\ t_{add\_local\_response\_norm} &= n_c \cdot x_{in} \cdot y_{in} \cdot z_{in}, \end{aligned} \tag{2.18}$$

without considering the expensive $x_{in} \cdot y_{in} \cdot z_{in}$ negative exponentiations. In this way, the complexity is no longer linear, being $O(n_c \cdot x_{in} \cdot y_{in} \cdot z_{in})$.

2.2.4 Linear or Fully-Connected Layer

Given an input of size x_{in} and a number of outputs (usually referred to as hidden neurons) x_o, the time complexity is defined as follows

$$\begin{aligned} t_{mul\_fc} &= x_{in} \cdot x_o, \\ t_{add\_conv} &= (x_{in} - 1) \cdot x_o, \end{aligned} \tag{2.19}$$

with a complexity $O(x_{in} \cdot x_o)$.

2.3 Energy Consumption of Common Operations

In the related literature, several works are aiming at estimating or modeling the energy consumption of every single instruction or by a sequence of instructions (e.g., to compute the energy required by a given algorithm) (Castillo et al., 2007; Fei et al., 2004; Horowitz, 2014; Molka et al., 2010; Tiwari et al., 1996). A brief review and some actual results follow. The simplest, used for instance in (Tiwari et al., 1996), estimates the energy of a single instruction *op* in a classic way by multiplying the power by the time the instruction requires:

$$E_{op} = P_{op} \cdot t_{op} = V_{cc} \cdot i \cdot N_{op} \cdot \tau, \tag{2.20}$$

where V_{cc} is the supply voltage, i the average current, and $N_{op} \cdot \tau$ the execution time expressed as the product of the number of *op* clock cycles and the clock frequency τ.

The described basic model is then extended in several works. A few examples follow. At first, (Tiwari et al., 1996) takes into account the overhead due to operations switching (formally, the energy required to execute two different instruction is greater than the sum

Table 2.2: *The energy measurements, expressed in pJ, on different operations for different technologies and processors as stated in (Castillo et al., 2007; Horowitz, 2014; Molka et al., 2010).*

(a) *The energy measurements (Horowitz, 2014) on a 45 nm technology to execute the given operation only, i.e., without considering moving operands from register to the arithmetic unit and storing the results back.*

Operation	Energy (pJ)
8-bit Integer Add	0.03
32-bit Integer Add	0.1
16-bit Floating-Point Add	0.4
32-bit Floating-Point Add	0.9
8-bit Integer Multiply	0.2
32-bit Integer Multiply	3
16-bit Floating-Point Multiply	1
32-bit Floating-Point Multiply	4

(b) *The energy measurements in pJ (of the whole floating-point 32-bit instruction) on two different processors, the high-performance Intel Xeon X5670 from (Molka et al., 2010) and the old ARM9TDMI for microcontrollers from (Castillo et al., 2007). For both the processors, the nominal operating frequency f and tension V_{dd}, and the manufacturing technology are reported.*

Operation	ARM9TDMI $f = 200$MHz $V_{dd} = 2.5$V 250nm techn.	Intel Xeon X5670 $f = 2.93$GHz $V_{dd} = 1.35$V 32nm techn.
Add	2250	111
Multiply	2250	164
Comparison	3375	-

of the energy to execute the same two instructions independently). (Molka et al., 2010) defines different models to employ the characteristics and parameters of different kind of instructions. Finally, a few works focus on specific architectures, e.g., very long instruction word processors (Bona et al., 2005; Laurent et al., 2001).

Table 2.2 reports results specific to one particular processing technology and a few processors with different purposes as stated in (Castillo et al., 2007; Horowitz, 2014; Molka et al., 2010). More in detail, Table 2.2a reports the rough energy consumptions for a 45 nm technology to execute only the given operation, i.e., without considering moving operands from register to the arithmetic unit, storing the operation output back, and other memory management mechanisms (Horowitz, 2014). These results are as expected, showing that the floating-point operations are significantly energy-hungry than integer ones. The same is true for multiplications with respect to additions and higher precision data types w.r.t. lower precision ones (e.g., 32-bit integers w.r.t. 8-bit ones).

Table 2.2b compares the energy consumptions of the whole considered operations (with 32-bit floating-point data types) from operands gathering to output saving of two processors with different purposes: the ARM9TDMI, an old core designed for microcontrollers (Castillo et al., 2007); and the high-performance Intel Xeon X5670 designed for data-centers (Molka et al., 2010). The former has a 250 nm manufacturing technology and nominally operates at 200 MHz and 2.5 V. The latter is a 32nm core operating at 2.93 GHz and 1.35 V. As expected, the energy consumptions are significantly different for the two cores (the Intel Xeon requires about 5% of the energy of the ARM9 core), resulting in the need for application and technology-specific energy consumptions analysis in order to define the values of e_{mul} and e_{add} in Equation (2.5).

CHAPTER 3

Related Literature

The analysis of the related literature starts from the dedicated hardware solutions in order to support machine and deep learning algorithms (Section 3.1), then Section 3.2 presents the approximation techniques that aim at reducing either the memory footprint of deep learning algorithms by means of task dropping (Section 3.2.1) and precision scaling (Section 3.2.2) techniques, or their computational load by introducing Early-Exits within the deep learning models themselves (Section 3.2.3). Section 3.3 analyses the deep learning models in presence of concept drift. Section 3.4 presents the related literature about Tiny Machine Learning and, finally, Section 3.5 briefly discusses a few solutions to distribute the deep learning pipeline among (possibly heterogeneous) IoT units or embedded systems.[3]

3.1 Dedicated Hardware Solutions

This section (briefly) presents the research direction of custom hardware for machine and deep learning models. The resulting hardware solutions are characterized by the best power consumptions, inference time, and throughputs. The downside is the complexity of the design phase as well as the reduced or lack of flexibility.

In this research direction, the hardware units allowing for parallel processing and thus increasing the throughput of the DL processing while reducing the inference time have faced a massive growth in interest, such as the Graphical Processing Units (GPUs). Al-

[3]This analysis contains the papers listed in Section 1.2 and presented within this thesis.

though GPUs were originally dedicated to the rendering in computer graphics, they have rapidly become one of the most common choices in DL processing, with data centers equipped with clusters of GPUs (Cano, 2018; Cui et al., 2016; Gu et al., 2019; Wang et al., 2019b). Recently, Tensor Processing Units (TPUs) (Jouppi et al., 2018; Wang et al., 2019b) emerged as new technological hardware specifically meant for machine and deep learning processing on data centers. Then, Field Programmable Gate Arrays (FPGAs) are another common choice for DL processing due to their reconfiguration capabilities (Qiu et al., 2016; Véstias et al., 2017; Zhang et al., 2015a).

Despite GPUs, TPUs, and FPGAs provide significant increments in terms of throughput, their energy requirements are out of purpose for IoT units or embedded systems that are not equipped with any of them (Alippi et al., 2018; Wang et al., 2019b). Moreover, the design of custom processors, neural engines, and other optimized hardware solutions allow to achieve further improvements in the throughput and, at the same time, reduce the power consumption, with greater gains in all those scenarios involving quantization or custom data types (e.g., fixed-point representations) (Wang et al., 2019a). As an example, the Neural Processing Unit proposed by (Fowers et al., 2018) claims an order of magnitude improvement w.r.t. GPUs on large Recurrent Neural Networks (RNNs) models, whereas the 28-nm system-on-chip (SoC) specifically designed for IoT solutions presented in (Whatmough et al., 2018) guarantees the lowest energy consumption while preserving the accuracy of the employed DL model (on the MNIST dataset (LeCun et al., 1998) the energy required by a single inference is 0.36 μJ at 667 MHz and 0.57 μJ at 1.2 GHz). The last-mentioned solution is particularly interesting because it is one of the solutions specifically meant for IoT units (Whatmough et al., 2018), although the complexity of the design stage limits this research direction in the IoT world. A few examples of custom hardware solutions are the Intel Nervana (Hickmann et al., 2020), the IBM TrueNorth (Akopyan et al., 2015), the Microsoft BrainWave (Chung et al., 2018), as well as (Ando et al., 2017; Fowers et al., 2018; Knag et al., 2020; Moons et al., 2017; Shin et al., 2017).

(Guo et al., 2017b; Wang et al., 2019a) provide a survey on custom hardware solutions for deep learning models, as well as on accelerators to further increase the throughput without modifying the underlying hardware platform. A common characteristic to many custom hardware solutions is their specificity w.r.t. a group or a few families or machine or deep learning solutions. For instance, on FPGAs, (Blott et al., 2018; Gao et al., 2019; Han et al., 2017; Umuroglu et al., 2017; Wang et al., 2018b; Zhang et al., 2017) provided solutions for RNNs models, whereas (Boutros et al., 2018; Guo et al., 2017a; Wei et al., 2018; Venieris and Bouganis, 2018) focused on Convolutional Neural Networks (CNNs). (Wang et al., 2018b) proposed the C-LSTM (LSTM is a recurrent deep learning model) framework that computes the FPGA implementation for an LSTM through compression of the parameters, quantization of inputs and activations to 16-bit, and a few other optimizations for the inference. A similar approach is that of (Han et al., 2017) that employed both quantization and compression techniques to design LSTMs on FPGAs for speech recognition applications, with an implementation consuming (in terms of energy) less than 10% and being three times faster during inference than GPUs. In the case of CNNs, (Venieris and Bouganis, 2018) proposed the FPGA ConvNets framework that models the CNNs as a graph to compute the corresponding optimal hardware implementation (up to 1.05x more energy-efficient than GPUs), where the nodes are the optimized hardware implementations of some common CNN layers. Analogously, (Boutros et al., 2018) implemented FPGA-

specific digital signal processing blocks for CNNs working with low-precision computation (i.e., with 8 or 4-bit data formats), as well as (Guo et al., 2017a) that suggested a similar framework aiming at designing the best FPGA mapping of CNNs by relying only on 8-bit data types and being up to 2 times more energy efficient than the corresponding GPU implementation. Finally, (Blott et al., 2018; Umuroglu et al., 2017) proposed the FINN framework that focuses on the quantization of weights, including also mixed-precision solutions (i.e., where the layers of the pipeline can have different data precision formats).

Also, in the contest of custom hardware platforms, there are solutions designed for CNN processing (Chen et al., 2014; Farabet et al., 2011; Moons and Verhelst, 2016; Sharma et al., 2018), recurrent processing (Wang et al., 2017b), or both (Shin et al., 2017). (Wang et al., 2017b) designed a 90-nm solution specifically meant for LSTM, that can compress up to 95% of the original memory requirements (through some of the hardware optimization mentioned above for FPGAs) with a small impact on the accuracy. (Moons and Verhelst, 2016) designed a 40-nm solution exploiting the sparsity of convolutions as well as dynamic computation precision (i.e., the employed CNN computation can be done at 16, 8, or 4 according to desired energy consumption without affecting the inference time) in order to provide high throughput while optimizing the energy consumption in CNN processing. (Sharma et al., 2018) considered a 45-nm solution to present the Bit Fusion accelerator, that can offer dynamic precision in the deep learning models with a significant gain in terms of energy consumption w.r.t. to GPUs.

Other works tried to optimize some common routines in DL processing. (Dundar et al., 2016) was one of the first works in this field proposing to transform (in a compiler-way) the CNNs pipeline into a sequence of operations to be executed in a hardware accelerator with memory constraints. (Albericio et al., 2016; 2017) addressed the problem of reducing the number of useless operations (e.g., multiplications involving zero operands), whereas (Moss et al., 2018) designed matrix multiplication routines for FPGAs that are intensively used, for instance, in CNNs. Other works in this field can be found in (Kim et al., 2021; Lee et al., 2018; Long et al., 2018; Ueyoshi et al., 2018; Yuan et al., 2019).

A further step considers optimized architecture for binary or ternary computing. (Knag et al., 2020) designed a 10-nm CMOS accelerator for binary DL models' computation with reduced area and higher energy efficiency with respect to other binary solutions (Moons et al., 2018; Valavi et al., 2019). (Ando et al., 2017) instead designed an in-memory chip (i.e., without the need for external memory accesses) supporting both binary and ternary operations. (Andri et al., 2016) designed a 65nm solution specifically inspired by Binary-Connects CNNs (Courbariaux et al., 2015; Hubara et al., 2017). Other optimized hardware solutions for binary, ternary (or both of them) DL processing can be found in (Chen et al., 2018b; Conti et al., 2018; Kim et al., 2019; Shan et al., 2020; Whatmough et al., 2018; Yin et al., 2018).

3.2 Approximating Deep Learning Techniques

A survey on approximating techniques, not tailored to DL solutions, is available in (Mittal, 2016). Among them, the widely used approximation techniques in DL models are task dropping or skipping and precision scaling techniques. In the former approach, detailed in Section 3.2.1, the whole computation is reduced through skipping portions of the DL

models (e.g., pruning layers (Alippi et al., 2018; Han et al., 2016) or portions of them), with the interesting case of Early–Exit approaches (Bolukbasi et al., 2017; Disabato and Roveri, 2018), where the computation path within CNNs depends on the input information content (see Section 3.2.3 for details). In the latter approach, described in Section 3.2.2, either the parameters θ of a DL model $\mathcal{D}$ or the activations are approximated by relying on quantization techniques and different data types.

In addition to these techniques, a third approach designs techniques to spread the DL computation among many (possibly heterogeneous) IoT units or MCUs. Section 3.5 provides insights on such approaches.

3.2.1 Task Dropping Approximations

The related literature about task dropping introduces several different works that reduce the memory footprint (and, by removing tasks, also the computational load) of deep learning algorithms. These works can be grouped into pruning mechanisms (Alippi et al., 2018; Alvarez and Salzmann, 2017; Dong et al., 2017; Han et al., 2016; Li et al., 2017; Ullrich et al., 2017), novel optimized deep learning architectures (Chollet, 2017; Howard et al., 2017; Iandola et al., 2016; Liu et al., 2015; Lu et al., 2019; Yang et al., 2015; Zhang et al., 2018a), or optimizations of the deep learning processing pipeline (Chen et al., 2019b; Jaderberg et al., 2014; Li et al., 2019; Rigamonti et al., 2013).

The pruning algorithms can be grouped according to the "level" at which they operate. Structured algorithms (Alippi et al., 2018; He et al., 2017; Li et al., 2017; Luo et al., 2017) prune the structure of the deep learning models (e.g., the convolutional channels, or the layers), whereas unstructured ones can prune at parameters level (Han et al., 2016; Huang and Wang, 2018). (Han et al., 2016) significantly reduced the memory footprint of deep learning solutions through a three-stage pipeline. At first, the network pruning removes all the weights that, at the end of the DL model training, are below a given threshold (and then refines the resulting network). In a second step, the remaining weights are quantized (see Section 3.2.2 for the related literature) and clustered through k-Means, with all the weights within the same cluster sharing the value of the centroid. The third and last step considers Huffman coding (Van Leeuwen, 1976) to further reduce the memory footprint. As an example, the AlexNet CNN can be compressed from 240 to 7 MB with no drop in its classification accuracy. (Ullrich et al., 2017) modified the approach of (Han et al., 2016) by introducing a training procedure that in one step compresses the model with both quantization and parameters' sharing. Similarly, (He et al., 2017; 2018; Li et al., 2017; Luo et al., 2017) suggested pruning filters within convolutional layers to reduce the memory footprint. In this way, there is no need to develop novel processing solutions to deal with the compressed pipeline (e.g., to handle shared weights saved in common locations). (Dong et al., 2017) instead pruned parameters within the model according to the second-order derivative of the error function, proving bounds on the maximum accuracy drops. (Alvarez and Salzmann, 2017) suggested defining a training function that also encompasses the target of finding a compressed representation of weights by enforcing their representing matrices to have low-rank. Other pruning mechanisms can be found in (Adamczewski and Park, 2021; Carreira-Perpinán and Idelbayev, 2018; Gordon et al., 2018; Lin et al., 2020b; Liu et al., 2017; Tan and Motani, 2020; Yeom et al., 2021; Yu et al., 2018).

Interestingly, (Liu et al., 2019) observed that structured pruning algorithms from a larger model along with a fine-tuning of the pruned model can lead to worst or at most

comparable performances with respect to training the pruned model from scratch with random weights. The same results partially hold for unstructured pruning algorithms. Following this work, the pruning mechanisms should be considered only when a pre-trained model is already available, otherwise training a deep learning model designed to satisfy the technological constraints might result in a better solution.

The research in the direction of reducing the memory footprint of deep learning models has also produced novel models with architectural optimizations. (He et al., 2016; Yang et al., 2015) reduced the memory footprint of fully-connected layers by replacing them with an average pooling (He et al., 2016; Szegedy et al., 2015) or by introducing matrix factorization techniques in the fully-connected layers parameters (Yang et al., 2015). (Iandola et al., 2016) proposed the SqueezeNet, a CNN with less than 0.5 MB of memory footprint, but the same accuracy of the AlexNet one. The design choices to reduce the number of parameters and, in turn, the memory footprint are three: replace the 3x3 convolutional filters with 1x1 ones, introduce downsamples late in the architecture to have larger activation maps in the first convolutional layers, and the squeeze layers to reduce the number of inputs each convolutional filter covers. MobileNets (Howard et al., 2017) introduced the depth-wise convolutions that factorize standard convolutions as a single filter per each input channel and a 1x1 point-wise convolution to combine their outputs, with a significant drop in the required computational load. Other models employing depth-wise convolutions are the Xception (Chollet, 2017), the Factorized Networks (Wang et al., 2017a) or the ResNeXt (Xie et al., 2017). The ShuffleNet (Zhang et al., 2018a) further reduced the computational complexity of depth-wise convolutions and, in particular, of point-wise ones, by organizing the convolutional inputs into groups, each of them processed by a subset of convolutional filters (similar to the approach of AlexNet (Krizhevsky et al., 2012) whose goal was to split the computation among GPUs and to that of the NasNet (Zoph et al., 2018)). (Liu et al., 2015) proposed Sparse Convolutional Neural Networks, where the learning forces the majority of weights to be zero, zeroing more than 90% of the weights with a drop less than 1% in accuracy. Similarly, (Lu et al., 2019) proposed super sparse convolutional kernels where all the values except one are forced to be zero. (Molchanov et al., 2017) relied on Variational Dropout (Gal and Ghahramani, 2016) regularization (i.e., a way to see the dropout (Srivastava et al., 2014), i.e., the probability of skipping a neuron during training in dense layers, as a Bayesian regularization) to generate sparse deep learning models. For instance, the number of VGG parameters can be decreased by a factor of 68. Other works employing sparsity to reduce the memory footprint of deep learning models can be found in (Elsen et al., 2020; Huang and Wang, 2018; Li et al., 2020b; Quesada et al., 2018; Ren et al., 2018; Scardapane et al., 2017; Zhou et al., 2016a; Wen et al., 2016).

A similar approach aims at optimize the deep learning computational pipeline and, in particular, that of the convolutional layers. (Rigamonti et al., 2013) reduced the convolutions computation by defining the convolutional filters as linear combinations of smaller separable filters. A similar work is that of (Jaderberg et al., 2014) that drastically reduced the convolutional computation load by forcing the training to learn a set of low-rank convolutional filters. Other works suggested various approaches to reduce the convolutional complexity through various matrix factorization techniques (Ioannou et al., 2015; Lebedev et al., 2014; Li et al., 2019; Silva et al., 2017), orthogonal convolutions (Wang et al., 2020a), or other approaches (Denton et al., 2014; Zhang et al., 2015b).

Finally, in the field of natural language processing, (Li et al., 2020c) observed that the best results in defining compressed models derived from pruning of overparameterized ones, whereas (Ganesh et al., 2021) suggested reusing the techniques mentioned in this section for CNNs and other deep learning models. In such a research direction, examples of structured pruning solutions can be found in (Chi et al., 2021; Fan et al., 2020; Khetan and Karnin, 2020; Ganesh et al., 2021; Hou et al., 2020; Xu et al., 2020), whereas unstructured ones in (Chen et al., 2020; Gordon et al., 2020; Mao et al., 2020; Sanh et al., 2020).

3.2.2 Precision Scaling

Machine and deep learning models typically rely on 32-bit floating-point data types (Gupta et al., 2015). Precision scaling techniques to a reduced number of bits (e.g., 16, 8, or even 1) can significantly reduce the memory footprint of the considered deep learning solution. In this direction several works proposed novel data-type representations (Das et al., 2018; Köster et al., 2017) as well as fixed-point representations (Anwar et al., 2015; Han et al., 2016; Lin et al., 2016; Vanhoucke et al., 2011; Wang et al., 2018a).

(Köster et al., 2017) proposed a new hybrid data format in the middle between floating-point and fixed-point representation. As in floating-point there are N bits allocated for the mantissa and M for the exponent, but the exponent is kept common to all elements within a tensor and thus defined only once for each tensor. In particular, using a format with 16 bits for the mantissa and 5 bits for the common exponent, the authors achieved similar classification errors on state-of-the-art CNN models with 32-bit precision and outperformed corresponding 16-bit ones. Similarly, (Courbariaux et al., 2014) suggested keeping common and adaptive scaling factors (i.e., the equivalent of exponent part in floating-point) in groups of variables. Other examples of dynamic float point data types can be found in (Das et al., 2018; Johnson, 2018; Mellempudi et al., 2017). In contrast, works exploiting different floating-point representations are (Agrawal et al., 2019; Burgess et al., 2019; Cambier et al., 2020; Henry et al., 2019) with different configurations of the bits reserved to mantissa and exponent, e.g., the so-called Brain Floating Point 16 (BFloat 16) with 7 bits for the mantissa and 8 for the exponent.

(Gupta et al., 2015) studied the effects of introducing a 16-bit fixed-point representation in deep learning processing, proving a negligible or null drop in accuracy with their proposed stochastic quantization method. The fixed point representation is $\langle i, f \rangle$ (with i the bits for the integer part, f those for the fractional part, and $i + f = 16$ the number of bits used to represent each value) and can represent all the values from -2^{i-1} to $2^{i-1} - 2^{-f}$ with a step of 2^{-f}. Instead of quantizing each number to their corresponding nearest quantized value, (Gupta et al., 2015) solution assigns each number to the nearest quantized value with a probability that is as higher as these two values are close. (Jacob et al., 2018; Zhu et al., 2020), instead, developed a CNN framework that relies only on 8-bit integer values on both the inference and the training. Similarly, (Wang et al., 2018a) employs integer 8-bit computation, but with 32-bit floating-point gradients during training, whereas (Soudry et al., 2014) proposed a Bayesian approach to learn with 8-bit parameters. Other works employing 8-bit representations for deep learning parameters are (Banner et al., 2018; Dettmers, 2016; Wu et al., 2018; Yang et al., 2019), whereas works employing 4 to 6 bits representations can be found in (Elhoushi et al., 2021; Miyashita et al., 2016; Sun et al., 2020).

Literature about binary or ternary solutions exists with several variants on how and where the quantization is performed (Courbariaux et al., 2015; Hubara et al., 2017; Hwang and Sung, 2014; Kim and Smaragdis, 2018; Liu et al., 2018; Venkatesh et al., 2017; Wan et al., 2018). (Rastegari et al., 2016) proposed both Binary-Weight-Networks (BWNs) and XNOR-Networks, both based on CNNs. BWN binarizes only the parameters of the original CNN through the sign function, whereas an XNOR-Network applies the quantization to inputs, activations, and CNN's parameters, resulting in a speed-up (on CPUs) up to 65x w.r.t. 32-bit CNNs. Moreover, an architectural design rule has been devised to minimize quantization's impact on a quite common CNN block employing a convolutional, a batch-normalization, and a (max) pooling layer. The batch normalization is carried out prior to the binarization of the inputs so that their mean is zero. Then the pooling strictly follows the convolution. In this way, the equivalent XNOR version of the AlexNet (Krizhevsky et al., 2012) achieves the same accuracy on the ImageNet dataset. In a similar way, (Kim and Smaragdis, 2018) proposed Bitwise Neural Networks, that having both the parameters and the activations binary, replaced all the deep learning processing with bitwise operations that efficiently pack 32 original operations in a single instruction. (Wan et al., 2018) proposed optimized kernels for the so-called Ternary-Binary Networks (TBNs) having ternary inputs and binary parameters, that can guarantee a 32x drop in memory footprint and about 40x speed-up in inference time. (Courbariaux et al., 2015; Hubara et al., 2017) suggested two different quantization functions for binary networks, i.e., the sign function and a stochastic variant of the sign function, where the sign of x is computed on x minus a uniform variable z and the function maps each input $x - z$ to $+1$ with a probability equal to the hard sigmoid of x, to -1 otherwise. Moreover, (Courbariaux et al., 2015; Hubara et al., 2017) studied how to allow the backpropagation on these networks by relying on real-valued gradients of the weights, an adequately defined estimator of the gradient function that zeroes the gradient for large values and is based on the proposed stochastic sign function, and a shift-based implementation of the batch-normalization that provides significant speed-ups during the training. The resulting CNNs (or RNNs) behave closely to corresponding full-precision models in terms of evaluation metrics. Other works studying training on binary or ternary networks are (Hwang and Sung, 2014; Seide et al., 2014; Tang et al., 2017b; Zhou et al., 2016b).

Many works studied the optimal way to quantize the representation when employing fixed-point or fixed-value representations for deep learning computation (Anwar et al., 2015; Cai et al., 2017; Cai and Vasconcelos, 2020; Cai et al., 2020b; Chen et al., 2021; Gong et al., 2014; Hwang and Sung, 2014; Langroudi et al., 2021; Lin et al., 2016; Pouransari et al., 2020; Yang and Jin, 2021; Yao et al., 2021), whereas a conceptually similar approach studies how to vary the number of bits used to represents both the deep learning parameters and the activations (Bulat and Tzimiropoulos, 2021; Jin et al., 2020). (Anwar et al., 2015) introduced an optimization problem to find out the best fixed-point representation for each layer of a given CNN. Such an optimization problem finds out the best quantization configuration for each layer while keeping all the remaining ones at the original 32-bit floating-point precision. At the end, all the layers are quantized to their best fixed-point representation, with a mean gain of 10% on the memory footprint. Similarly, (Hwang and Sung, 2014) designed an optimization problem that initializes three possible quantized values (-x, 0, and +x, then encoded as -1, 0, and +1) through the minimization of the squared quantization error and then optimizes them via an exhaustive

search. (Lin et al., 2016) proposed several quantizers with different step-sizes in order to minimize the quantization error, either symmetric or not to include the zero. For instance, a symmetric 1-bit uniform quantizer encodes values -0.5 and 0.5 (step-size of 1), a symmetric 2-bit quantizer for Normal inputs encodes the values -1.494, -0.498, 0.498, and 1.494 (step-size of 0.996), whereas its corresponding asymmetric quantizer the values -0.996, 0.0, 0.996, and 1.992. In particular, (Lin et al., 2016) also devised an optimization problem to design different quantizations for each CNN layer that tries to minimize both the CNN's memory footprint and the introduced quantization error. A different approach to quantization is that of (Gong et al., 2014), where the CNN's parameters are quantized by k-means clustering of their values (k corresponds to the power of two of the desired number of bits in the compressed representation), with the centroids of the clustering algorithm being the quantized values. Recently, (Chen et al., 2021) designed a discrete optimization problem to address the mixed-precision quantization problem, i.e., when each layer of a CNN can be quantized with a different number of bits.

Finally, in the field of neural language processing, (Bhandare et al., 2019; Lin et al., 2020c; Zafrir et al., 2019) introduced quantization to 8-bit on BERT models (Devlin et al., 2019). (Shen et al., 2020) studied quantized versions of BERT models, and through the second-order Hessian information derived, they designed 2-bit models with less than 2% drop in accuracy, whereas (Bai et al., 2020) suggested a binary BERT model.

3.2.3 Early–Exit DL Models

The term *Early–Exit* DL models refers to all the models whose processing pipeline strictly depends on the information content brought by the input, i.e., the computation can be carried out at given intermediate checkpoints within the DL models, avoiding the computation of the remaining layers. Differently from the above-mentioned task dropping techniques, here the computational gain derived from the introduced approximations is usually not fixed, being conditioned on when the computation ends up at each checkpoint. Needless to say, these solutions can be easily distributed over a group of devices by assigning to each one the computation up to a given checkpoint, as suggested by (Disabato et al., 2021b; Scardapane et al., 2020) with different approaches.

Almost all the works in this field focus on CNNs (Bolukbasi et al., 2017; Disabato and Roveri, 2018; Kaya et al., 2019; Passalis et al., 2020; Teerapittayanon et al., 2016; Xu et al., 2018; Yang et al., 2020; Zhou et al., 2019). Inspired by the internal classifier of the Inception V3 (Szegedy et al., 2015) (in that CNN used to speed-up the training phase assuming that it predicts as the final one), (Kaya et al., 2019) modifies existing CNNs by introducing one or more internal classifiers. There is the possibility to train the internal classifiers only (e.g., when attached to pretrained CNNs) or to train the whole architecture. In the latter case, the predictions and the importance of the internal classifiers are weighted with configurable coefficients. The authors also define the concept of *overthinking*, which is the case in which the CNN is able to predict the correct label at an internal classifier. To mitigate this problem, they define a confidence threshold the internal classifier should reach to provide the final classification (and exits). (Disabato and Roveri, 2018) proposes Gate-Classification CNNs that are characterized by the presence of one or more intermediate classifiers, namely Gate-Classification layers, that are able to provide the final classification about the input if they have enough confidence on, skipping all the subsequent layers. The training of these CNNs first optimizes all its parameters θ (similarly to (Kaya

et al., 2019)), then learns a confidence threshold (for all the intermediate Gate-Classifiers) aiming at not reducing the whole architecture accuracy. The BranchyNet (Teerapittayanon et al., 2016) follows the same idea, but the authors develop a one-step training procedure optimizing the classifiers and their decision functions jointly.

Similarly, (Bolukbasi et al., 2017) defines an adaptive early–exit strategy that introduces decision functions after several DL layers, that allows to skip the remaining computation if they can provide the final classification. These decision functions are learned along with the parameters of the model in order to keep the mean computational budget (across several inferences) under a given application-specific threshold. Following the approach proposed in (Wang et al., 2015), (Bolukbasi et al., 2017) suggests a second solution having several DL models into a direct acyclic graph. The choice of the path within the graph to provide a prediction on a given input depends on the available computational budget (i.e., the path is chosen before the computation itself).

(Yang et al., 2020) proposes Resolution Adaptive CNNs, where multiple CNNs of different spatial resolutions are adopted. For each input sample, the inner CNNs are processed from the lowest to the highest resolution, with the possibility to provide the final classification after any of them if enough confidence on the final prediction is achieved.

A different approach to the problem is that of SkipNet (Wang et al., 2018c), where the CNN architecture is modified by the introduction of special (gating) residual connections aiming at deciding whether or not executing the next layer (or group of layers), based on the output of the previous one. The authors suggest a hybrid algorithm combining supervised information and a reinforcement learning approach to allow these gates to learn the "skipping" policy. Such an approach is able to balance the final accuracy and the required mean amount of computational gain. Similarly, (Verelst and Tuytelaars, 2020) suggests convolutional layers where the convolutional filter is not applied to the whole input, but only on the regions of interest, according to the input information. Although both the proposed approaches are interesting and allow to obtain significant computational gains in some tasks, they might not match the memory constraints imposed by IoT devices or MCUs, requiring to have the whole model available for every inference.

Finally, (Zhou et al., 2020) proposes the Patience-based Early Exit model that enriches the (pre-trained) ALBERT model (Lan et al., 2019) for NLP, with early exits. The author introduces a classifier after each layer of the pretrained language model and provides the final classification if for a given number of consecutive layers the final prediction is unchanged (this number is called *patience*).

3.3 Machine and Deep Learning in Presence of Concept Drift

The literature about machine and deep learning in presence of concept drift refers to adaptive solutions able to deal with concept drift affecting the data generating process. The related literature usually groups them into two main families: passive and active (Ditzler et al., 2015; Gama et al., 2014; Lu et al., 2018).

Passive solutions adapt the model at each incoming data, disregarding the fact that a concept drift has occurred in the data-generating process (or not). The gradual forgetting classifiers, e.g. (Elwell and Polikar, 2011; Krawczyk and Woźniak, 2015; Polikar et al., 2001), which reduce the importance of older samples over time, are examples of passive solutions. The Concept Drift Very Fast Decision Tree (CDVFDT) (Hulten et al., 2001)

introduces a Decision Tree that learns new subtrees on incoming data. However, most passive solutions employ ensemble methods and their adaptation mechanisms consist in adding, removing, or weighting the ensemble base classifiers, e.g., Streaming Ensemble Algorithm (Street and Kim, 2001), Dynamic Classifier Selection (Almeida et al., 2018), or the adaptive ensemble of Decision Trees proposed in (Pietruczuk et al., 2017). Deep learning-based passive solutions (Li et al., 2020a; Parisi et al., 2019; Pérez-Sánchez et al., 2018; Zenke et al., 2017) has to deal with the catastrophic forgetting phenomena, i.e., the propensity to discard the older knowledge during learning of new incoming information (French, 1999; Ratcliff, 1990). (Zenke et al., 2017) introduced synapses similar to biological ones, whereas (Li et al., 2020a) proposed a Bayesian approach encompassing a Gaussian Mixture Model to maintain the older information while learning the new one.

On the contrary, *active solutions* aim at detecting concept drift in the data generation process and, only in that case, they adapt their model to the new conditions. Change Detection Tests (CDT) are statistical techniques meant to sequentially process the incoming data inspecting for concept drift. (Ditzler and Polikar, 2011) proposed to use the Hellinger distance between the reference probability distribution and the one estimated on incoming data along with a t-test to detect changes. (Dasu et al., 2006) relies on bootstrapping several windows of data and the Kullback Leibler divergence as a measure of the distance among them. A few works detect changes with density estimation techniques (Bu et al., 2016; Duda et al., 2018). Other examples of CDT used in active solutions can be found in (Baena-Garcıa et al., 2006; Boracchi et al., 2018; Page, 1954; Wang et al., 2020b; Zambon et al., 2018). In active solutions, the adaptation stage following a concept drift detection is usually carried out in two steps (Alippi et al., 2013): first, the time instant the concept drift occurred is estimated by ad-hoc mechanisms (e.g., by Change-Point Methods); second, the obsolete knowledge, i.e., that acquired before the concept drift occurred, is discarded. To achieve this goal, the adaptation mechanisms typically rely on a window over the last acquired data, whose size is usually optimized over time to reduce its memory requirements (Aggarwal, 2006; Vitter, 1985), or on all the samples seen so far (suitably weighted) (Klinkenberg, 2004). Finally, deep learning-based active approaches (integrating deep learning solutions with active adaptive solutions) can be found in (Disabato and Roveri, 2019; Yang et al., 2019). (Disabato and Roveri, 2019) proposed a non-parametric CUSUM (Page, 1954; Lorden et al., 1971) that monitors the CNN classification error to inspect for changes. Once a change is detected, the adaptation employs the exploration of the CNN processing pipeline to identify the layers that have become obsolete due to the concept drift, which are then adapted. (Yang et al., 2019) proposed to rely on two different models operating in parallel, one of them continuously trained on novel incoming data. A change can be detected when the dissimilarity between the two models, i.e., the stable on older knowledge and the responsive one, overcomes a threshold.

3.4 Tiny Machine Learning Related Literature

Tiny Machine Learning (TML) is a relatively new research area aiming at designing machine and deep learning solutions that can be executed mainly on microcontrollers units (but also on IoT units) with memory constraints in the order of kilobytes and expected energy consumptions of milli- to micro-Watts. Prior to TML, there were several works aiming at reducing the complexity in memory, computation, and energy of deep learn-

ing solutions, but few of them with the target of IoT units or embedded systems (Alippi et al., 2018; Dundar et al., 2016; Kumar et al., 2017; Miyashita et al., 2016). In particular, (Alippi et al., 2018) proposed a methodology that takes into account the deep learning model to be executed on an embedded system along with the technological constraints on memory and computation such an embedded system introduces. After that, the methodology produces in output a set of approximated solutions by means of task dropping and precision scaling mechanisms that can be mapped on the considered embedded system and do not pareto dominate each other on an evaluation set. (Kumar et al., 2017) presented the Bonsai tree algorithm to enable the deep learning inference on IoT devices with 2 KB of RAM. Bonsai learns a sparse tree representation of the original deep learning model into a low-dimensional space where the points are projected (the parameters of the projection are learned along with the tree itself). The nodes within the learned tree are non-linear decision functions and the predictions of all the leaves are summed up to define the final model prediction. After the birth of TML, all the literature about deep learning approximations, here presented in Sections 3.1–3.2.3, has been taken into account to satisfy the severe technological constraints on memory, computation, and energy characterizing IoT units and embedded systems (Banbury et al., 2020; Disabato and Roveri, 2020; 2021; Sanchez-Iborra and Skarmeta, 2020).

As regards the target of the approximation mechanisms, most of TML literature focuses on approximated Convolutional Neural Networks (Banbury et al., 2021; Fedorov et al., 2019; Gopinath et al., 2019; Liberis et al., 2021; Lin et al., 2020a), with a few works considering recurrent DL architectures (Fedorov et al., 2020; Kumar et al., 2020; Venzke et al., 2020) and, to the best of our knowledge, no work addressing Tiny NLP. (Lin et al., 2020a) proposed a framework that explores several deep learning architectures under the desired technological constraints on memory, computation, and energy. Such optimization takes into account different layers as well different implementations of those layers in order to increase the variety of explored solutions as well as find out the best possible implementation for the produced model (in terms of layer implementations but also memory reuse whenever possible). Similarly, (Banbury et al., 2021) developed a framework that explores different architectures under the technological constraints of target micro-controller units by optimizing the number of operations. Other works exploring CNNs architecture for micro-controller units are (Fedorov et al., 2019; Liberis et al., 2021). (Gopinath et al., 2019) instead designed a compiler that translates machine and deep learning models into corresponding models with fixed-point representation in order to satisfy the given memory constraint. (Rusci et al., 2020) studied the impact of quantized networks in TinyML embedded systems with solutions employing representations with 2, 4, or 8 bits. Then, (Kusupati et al., 2018) developed recurrent models of 1 to 6 KB, namely FastGRNN (Fast, Accurate, Stable and Tiny Gated Recurrent Neural Network), that in addition to the standard update of the hidden state has a residual component from the previous state with only two additional scalar parameters added to the model, i.e., the weights of this residual component and the other component.

All the works mentioned above addressed the problem of enabling the inference of tiny machine or deep learning algorithms. However, in the literature, there are very few works proposing on-device learning mechanisms, i.e., where the training of tiny machine or deep learning algorithms can be carried out directly on the embedded system, the IoT or microcontroller unit is deployed on (Cai et al., 2020a; Disabato and Roveri, 2020; 2021; Ren

et al., 2021). (Cai et al., 2020a) presented the Tiny-Transfer-Learning (TinyTL) solution that enables the learning of the biases of a deep learning model, whereas the weights are frozen. In this way, the learning algorithm does not need to store the intermediate activations in order to back-propagate the gradients to the weights. Moreover, to compensate for the reduced learning ability derived from having frozen weights, TinyTL introduced a lite residual learning module in each layer: the output of each layer is then computed on the (frozen) weights and the learnable biases plus a residual component computed on a reduced version of the previous layer's activations with learnable weights. This lite residual component operates on smaller activations, thus has a significantly smaller impact on the activations stored by the backpropagation algorithm. However, the solution is not meant for micro-controller units. (Disabato and Roveri, 2020), instead, addressed the problem of learning on micro-controller units by relying on a deep learning-based feature extractor (designed by means of task dropping techniques to satisfy the technological constraints of the considered micro-controller unit) and a k-Nearest Neighbor classifier operating on the extracted features. Such TML solution can learn incrementally since the kNN is a non-parametric classifier whose adaptation consists in changes of its knowledge set (i.e., the set of saved samples used to classify a novel unseen one). Hence novel knowledge can be provided by simply saving incoming (supervised) samples within the kNN knowledge set. Such an approach is quite simple, has a relatively negligible adaptation cost, and can operate in all those scenarios where initially a very few supervised samples are provided to the TML algorithm. (Disabato and Roveri, 2021) extended the work of (Disabato and Roveri, 2020) with the introduction of three different adaptation mechanisms to make the model adaptive to concept drift. In this way, the deployed TML model can adapt its knowledge set by either discarding obsolete information (e.g., sample referring to the environment conditions previous to a concept drift) or introducing novel data. Finally, (Ren et al., 2021) proposed an online learning approach that can adapt deep learning models by introducing an extra learnable output layer, whereas the parameters of the original model are kept frozen.

3.5 Distributing the DL Computation on Pervasive Systems

Section 3.2 presented the related literature about approximation techniques to reduce the memory and computational requirements of machine and deep learning techniques, most of them then used in Tiny Machine Learning solutions (see Section 3.4). A totally different approach to the problem of reducing the memory, computation, and energy requirements of machine and deep learning solutions in order to match the corresponding constraints on memory, computation, and energy of IoT units, embedded systems, and micro-controller units, is that of distributing the deep learning pipeline among several units that can also be heterogeneous.

This idea derives from the offloading techniques for distributed computing systems. Here, the goal is not to reduce deep learning solutions' complexity but to move computationally-intensive processing to high-performing units of the distributed system. In this domain, (Shi et al., 2012) proposed a framework to optimally offload code in a pervasive system comprising mobile units with intermittent connectivity. Similarly, (Bozorgchenani et al., 2020; Kao et al., 2017; Pu et al., 2016) proposed frameworks to offload the code in resource-constrained mobile devices with the goal of reducing the total latency to carry out

the given task, but at the same time being aware of the energy status of the devices, i.e., the frameworks have as an additional goal to increase the system lifetime and to offload the computation primarily to those units with the highest energy. In particular, (Bozorgchenani et al., 2020) employs the possibility for the connected devices to harvest energy through solar panels with configurable parameters (e.g., size, inclination, position in the world, presence of obstacles, kind of nearby terrain, and so on). (Wang et al., 2020b) proposed an online-learning based scheduler to optimally distribute the computation in industrial IoT systems by taking into account the power status of the devices as well as the connectivity issues. Differently, (Hong et al., 2013) proposed a high-level programming language to design applications meant for Fog-Computing Sensor Networks able to hide the heterogeneity of computing nodes and their position in space. (Hu and Krishnamachari, 2019) proposed a low-complexity scheduler that increases the throughput in IoT clusters by relying on tasks duplication and splitting by taking into account communication and computation capabilities, but not IoT units memory constraints. Other similar works can be found in (Cuervo et al., 2010; Ghosh et al., 2019; Shi et al., 2014; Wu et al., 2021).

Very few works present in the literature encompass the code offloading of machine learning-based applications in pervasive systems, e.g., (Cheng et al., 2015) where the classification/pattern recognition tasks of a wearable device are partially offloaded to other computing units (e.g., mobile phones). In the field of deep learning solutions, (Kang et al., 2017) highlighted that a distribution of the deep learning model layers among the available mobile devices and the Cloud has significant advantages in terms of both latency and energy w.r.t. sending the acquired data to the Cloud and waiting back for the results. The study covers various applications, from image classification to natural language processing. In this direction, (Teerapittayanon et al., 2017) introduced a distributed framework for CNNs operating in edge computing platforms, with the possibility of distributing the CNNs computation, mostly restricted to the Cloud, also to edge or local devices. (Disabato et al., 2021b) proposed a quadratic optimization problem to distribute the computation of one or more deep learning models (even with Early-Exits or shared layers) into a heterogeneous network of IoT units by taking into account the constraints on memory and computation such IoT units introduce. (Zhao et al., 2018) instead proposed a framework that distributes the layers of the considered CNN models vertically, i.e., the computation of each single layer can be split (whenever possible, e.g., for convolutional filters) to also exploit the parallelism in computation and optimize the data reuse. (Hu and Krishnamachari, 2020) addressed the problem of transmitting the deep learning layers' activations among the IoT units processing consecutive layers by compressing them with autoencoder models. Other works aiming at distribute the deep learning computation are (Bhardwaj et al., 2019; Chen et al., 2019a; He et al., 2020; Laskaridis et al., 2020; Stahl et al., 2019; Tao and Li, 2018).

CHAPTER 4

Problem Formulation

This chapter formalizes the Tiny Machine Learning problem of executing (distributed) DL solutions on embedded systems, IoT units, or microcontrollers units. In particular, to simplify the description, the term IoT unit (or system) is used as a general term and thus employs all the different types of devices (i.e., also embedded systems and microcontrollers units). Section 4.1 details the data-generation process in both stationary and nonstationary environments. After that, Section 4.2 presents the concept of Deep Learning Model (DLM), whereas Section 4.3 provides the formalization of a network of heterogeneous IoT units on which it is possible to spread the computation of DLMs. Finally, Section 4.4 summarizes all the notation.

4.1 Data-Generating Process

Let $\mathcal{P}$ be a data generating process that, at each time instant t, provides a pair (x_t, y_t) sampled from an unknown probability distribution $p_t(x, y)$, where x is the input of the proposed solution $\mathcal{D}$ (e.g., an image or an audio clip) and $y \in \Delta$ its classification label.[4]

Moreover, following a *test-then-train* approach (Ditzler et al., 2015), the proposed solution $\mathcal{D}$ receives the supervised information (the true label y_t) only after it provides the classification output $\hat{y}_t = \mathcal{D}(x_t)$ on input x_t, at each time instant t. Without any loss of generality, the supervised information might not be provided at every time instant. In those cases, the proposed solution only provides its classification output.

[4]Let δ be the cardinality of Δ, i.e., the number of classes in the considered classification problem.

In a concept drift scenario, the process $\mathcal{P}$ might evolve over time, hence assuming that a shift in the distribution $p_t(x, y)$ might occur at an unknown time instant t^*. It is worth noting that the change in $p_t(x, y)$ might affect the input x (e.g., by the introduction of noise), the set Δ (e.g., class change), or both.

4.2 Deep Learning Model

In the most general way, a Deep Learning Model (DLM) can be split into M layers. Given an input x (e.g., an image or an audio waveform), the DLM $\mathcal{D}$ provides in output a value y (e.g., a label in a classification task or a value in a regression algorithm), i.e., $y = \mathcal{D}(x)$. More specifically, the processing pipeline within the DLM algorithm follows the sequence of layers (with a few exceptions, e.g., Early-Exit DLMs presented in Section 3.2.3), that is,

$$\begin{aligned} y &= \mathcal{D}(x) = \varphi_{\theta_M}^{(M)}(x_{M-1}) \\ x_\ell &= \varphi_{\theta_\ell}^{(\ell)}(x_{\ell-1}), \text{ for each } \ell \in \{1, \ldots, M\} \\ x_1 &= \varphi_{\theta_1}^{(1)}(x_1) \end{aligned} \tag{4.1}$$

where ℓ represents a generic layer, $\varphi_{\theta_\ell}^{(\ell)}$, for each $\ell \in \{1, \ldots, M\}$, is the function with parameters θ_ℓ accounting for the ℓ-th layer of the DLM, and x_ℓ the output (or intermediate activation) of that layer to be processed by the subsequent $\ell + 1$-th layer. The function φ that models the DLM layers can be, for instance, a convolutional layer, a non-linearity, a pooling operator, to name a few examples.

In particular scenarios, it is helpful to group a bunch of DLM layers according to their function. To provide a comprehensive formalization, a group of additional operators is defined:

- the feature extractor operator ϱ, that usually encompasses a subset of a DLM layers, with the goal of extracting features ζ from an input x, i.e., $\zeta = \varrho(x)$.
- the dimensionality reduction operator ς, that might be based on a DLM or a machine learning technique (such as Principal Component Analysis, Variance Thresholding, to name a few), with the goal of reducing the dimensionality of its input. Given an input x, the output $y = \varsigma(x)$ of the dimensionality reduction operator has $|y| \leq |x|$, with $|\cdot|$ cardinality operator.
- a classifier operator $\mathcal{K}$, that might be based on a DLM (e.g., a sequence of fully-connected layers) or on a machine learning technique (e.g., a k-Nearest Neighbor (Altman, 1992) or a Support Vector Machine (Cortes and Vapnik, 1995)), with the goal of providing the classification output. Given an input x, that might be either a "simple" input (e.g., images or audio) or the output of other DLM layers or operators, the classifier operator $\mathcal{K}$ provides the classification output y as $y = \mathcal{K}(x)$.

For instance, when considering Convolutional Neural Networks –DLM algorithms mainly used in image or video applications– their pipeline of layers can be organized into two operators: a feature extractor encompassing all the convolutional layers along with other related ones (e.g., non-linear, pooling, or batch normalization layers), then a classifier operator encompassing fully-connected layers providing the final classification output y of the input image x, i.e., $y = \mathcal{D}(x) = \mathcal{K} \circ \varrho(x)$.

4.3 A Possibly Heterogeneous IoT System

The IoT system comprises a set of C data-acquisition units $\{s_1, \ldots, s_C\}$ endowed with sensors and providing the inputs to be processed by the C DLMs (each DLM processes data coming from one data acquisition unit), a set $\mathbb{N}_N = \{1, \ldots, N\}$ of N possibly heterogeneous IoT units implementing code execution, and one target unit f receiving all the C DLMs' outcomes to make a decision or activate a reaction.[5] Without any loss of generality, the C data-acquisition units are assumed only to acquire data and do not participate in the computation. The i-th IoT unit $i \in \mathbb{N}_N$ is characterized by its own constraints on maximum memory capacity $\bar{m}_i$ and available computation $\bar{c}_i$.

The IoT system can be modelled as a graph $G(V, E)$ of nodes V and arcs E. The nodes set includes the IoT units, the sources, and the target unit f, i.e., $V = \{\mathbb{N}_N \cup \{s_1, \ldots, s_C, f\}\}$. An arc e_{i_1,i_2} between unit i_1 and i_2 exists in E if i_2 is within the range of the transmission technology the IoT unit i_1 is equipped with.[6] Let d_{i_1,i_2}, for each $i_1, i_2 \in V$, be the *hop-distance* between units i_1 and i_2 of V defined as the number of hops (communication steps), data need to take to reach i_2 from i_1. In other terms, distance d_{i_1,i_2} is the shortest communication path between units i_1 and i_2 within the graph G. Following the definition of shortest path in a graph, if no path between unit i_1 and i_2 exists, then $d_{i_1,i_2} = +\infty$. We also assume that no isolated node exists, i.e., $d_{i_1,i_2} < \infty$, for each $i_1, i_2 \in V$.

Let M_u, for each $u \in \mathbb{N}_C = \{1, \ldots, C\}$, be the number of layers characterizing the u-th DLM. Each layer j of DLM u, for each $j \in \{1, \ldots, M_u\}$ and $u \in \mathbb{N}_C$, is characterized by a given memory demand $m_{u,j}$ and computation $c_{u,j}$. More specifically, the memory complexity $m_{u,j}$ (in Bytes) defines the number of weights that layer j of DLM u has to store multiplied by the size of the data type used to represent those parameters (typically the 32-bit floating-point type). In contrast, the computational load $c_{u,j}$ measures the number of multiplications to be executed by that layer (see Section 2). Let $K_{u,j}$, for each $u \in \mathbb{N}_C$ and $j \in \{1, \ldots, M_u\}$, be the memory occupation of the intermediate representation transmitted from layer j to the subsequent layer $j + 1$ of DLM u, and $K_{u,s}$ and K_{u,M_u} be the memory occupation of the input of DLM u transmitted from source s_u to the unit executing the first layer of the u-th DLM and the final classification provided by layer M_u sent to the target unit f, respectively. In particular, K_{u,M_u} is either the classification label or the posterior probability of the classes resulting from the softmax layer.

4.4 List of Symbols

This section recaps all the nomenclature and symbols used within this document. It is crucial to point out that a few symbols overlap, but according to the Section or the context should be clear identify the correct meaning of them. For instance, the lowercase f is used to represent a frequency when the audio is employed, but also the IoT unit the output of

[5] The inputs are assumed to be generated by the data-generating process $\mathcal{P}$, either in stationary or nonstationary conditions.

[6] All the units in V are assumed to share the same transmission technology with a fixed transmission data-rate. If the IoT units i_1 and i_2 adopt two different transmission technologies such that i_2 is within the transmission range of i_1, but i_1 is not inside the one of i_2, then $d_{i_1,i_2} = 1 \neq d_{i_2,i_1}$ (loss of symmetry property).

a deep learning model is sent to when describing the proposed solution to distribute the computation of a deep learning model on a network of IoT units. Moreover, the lowercase x usually refers to an input, with its variant representing its possible transformations during a preprocessing step. Similarly, the lowercase y refers to a generic output of a deep learning model.

Data-Generation Process

Δ The set of classes in a classification problem.

δ The number of classes in a classification problem, i.e., the cardinality of Δ.

$\mathcal{P}$ The data-generation process.

x or x_t The input (at time t).

y or y_t The supervised information, e.g. the classification true label, of the input x (at time t).

Deep Learning Model

ℓ or j The index of layers within a DLM.

$\mathcal{D}$ A Deep Learning Model (DLM).

$\mathcal{K}$ A classifier operator.

θ DLM's parameters. The subscript ℓ indexes the layers of DLM.

$\varphi_{\theta_\ell}^{(\ell)}$ The function with parameters θ_ℓ representing the ℓ-th layer of a DLM.

ϱ A feature extractor operator.

ς A dimensionality reduction operator.

x_ℓ The output (activations) of the DLM's layer ℓ.

Complexity of Deep Learning Model

$\bar{c}$ Available computation of an IoT unit. A subscript further specifies the IoT unit this value refers to.

$\bar{e}$ Energy budget of an IoT unit. A subscript further specifies the IoT unit this value refers to.

$\bar{m}$ Memory capacity of an IoT unit. A subscript further specifies the IoT unit this value refers to.

c Computational or time complexity. A subscript further specifies what this complexity refers to (e.g., a layer, a deep learning model, and so on).

e Energy complexity. A subscript further specifies what this complexity refers to.

K Memory complexity of the activations of a deep learning solution at a given intermediate layer, at the input, or at the final layer . A subscript further specifies what this complexity refers to.

m Memory complexity. A subscript further specifies what this complexity refers to.

t_i Computational complexity expressed as inference time. A superscript further specifies what this complexity refers to.

IoT System

ω The minimum number of time a DLM model has to be processed in the IoT system. A subscript indexes the model.

ξ^{i_1,i_2} The probability that a transmission between the IoT units i_1 and i_2 fails. Alternatively, it can be seen as the retransmission rate.

C The number of deep learning models to be run in the IoT System.

f_u The u-th target unit (i.e., the node the final classification is sent to) in the IoT system.

$g_{u,j}$ The probability the final classification is provided at the layer j of the u-th deep learning model in the IoT system.

M The number of layers in each deep learning model to be run in the IoT System. A subscript indexes the model.

N The number of (heterogeneous) nodes in the IoT System.

$p_{u,j}$ The probability of executing the layer j of the u-th deep learning model in the IoT system.

s_u The u-th source (i.e. a node acquiring data) in the IoT system.

Deep Tiny Machine Learning

γ_ℓ The confidence threshold of a Gate-Classification layer (at layer ℓ of the DLM).

$\hat{\mathcal{D}}$ The approximated version of $\mathcal{D}$.

$\langle \hat{\alpha}_{\hat{\mathcal{D}}}, m_{\hat{\mathcal{D}}}, c_{\hat{\mathcal{D}}} \rangle$ The triple (relative to model $\hat{\mathcal{D}}$) of classification metric $\hat{\alpha}$ and memory m and computational c complexities used to define the pareto front $\mathcal{F}$ by $\mathcal{M}$.

$\mathcal{F}$ The pareto front of Deep TML models provided by $\mathcal{M}$.

$\mathcal{M}$ The methodology providing Deep Tiny Machine Learning Models.

$\tilde{\theta}$ The DLM parameters approximated by precision scaling (with parameter $\tilde{q}$).

$\tilde{m}$ The number of layers within $\hat{\mathcal{D}}$.

$\tilde{q}$ The precision scaling format within $\hat{\mathcal{D}}$.

Adaptive Deep Tiny Machine Learning

ϵ_t	The classification error at time t, i.e., 1 if $\hat{y_t} \neq y_t$, 0 otherwise.
$\hat{t}$	The CDT detection time.
$\hat{y_t}$	The output of the DLM $\mathcal{D}$ solution at time t (i.e., $\hat{y_t} = \mathcal{D}(x_t)$).
$\mathcal{L}$	The set of layers within $\mathcal{D}$ considered by pipeline exploration.
$\mathcal{T}_o, \mathcal{T}_n$	The two sets of old (o) and novel (n) samples used during adaptation.
$\tilde{\ell}$	The first obsolete layer (output of the pipeline exploration method).
ϖ	The size of the history window.
ϑ	The CDT decision function.
Ξ	The (minimum) size of $\mathcal{T}_o$ and $\mathcal{T}_n$.
p_e	The pipeline exploration method.
p_v	A p-value, output of the paired t-test.
t_r	The Refinement Procedure estimated change detection time.
x_t	The input provided by the data generation process $\mathcal{P}$ at time t.
y_t	The classification output provided by the data generation process $\mathcal{P}$ at time t.

On-Device Deep Tiny Machine Learning

$\hat{y_t}$	The output of the on-device TML solution at time t (i.e., $\hat{y_t} = \mathcal{D}(x_t)$).
ι	A parameter tuning the slope of p_a.
$\mathcal{K}$	In this domain, the classifier is a kNN.
$\mathcal{T}$	The training or knowledge set.
$\tilde{\upsilon_0}$	The exponential mean estimate of the CIT algorithm accuracy.
υ, n	In Active or Hybrid Tiny kNN algorithms, the parameters of the Binomial distribution used in the CUSUM CDT. More in details, υ_0 is the current estimated value, whereas υ_1 the (possibly unknown) value after a change.
Υ_1	In Active or Hybrid Tiny kNN algorithms, the set of possible values for υ_1 after the change.
ϖ	The size of either the adaptation window of Active Tiny kNN algorithm or the training set $\mathcal{T}$ of Hybrid Tiny kNN.
ϱ	In this domain, the feature extractor is the first layer of the ResNet-18.
ς	In this domain, the dimensionality reduction operator is the filter selection within the feature extractor.

ϑ In Active or Hybrid Tiny kNN algorithms, it represents a CDT function.

ξ In Active or Hybrid Tiny kNN algorithms, the number of initial samples used to estimate the value of υ_0.

ζ_t In Active or Hybrid Tiny kNN algorithms, the outcome of the Binomail distribution considered in the CUSUM CDT at time t.

g_t In Active or Hybrid Tiny kNN algorithms, the output of CDT decision function ϑ at time t.

h In Active or Hybrid Tiny kNN algorithms, the threshold of CDT.

p The maximum number of samples employed by the CIT passive algorithm.

p_a The probability a misclassified sample is added to the knowledge set $\mathcal{T}$ in the CIT algorithm.

s_t In Active or Hybrid Tiny kNN algorithms, the log likelihood ratio at time t. This quantity is an input of the CDT function ϑ.

t_r In Active or Hybrid Tiny kNN algorithms, the estimation of the change time.

W The adaptation window of Active Tiny kNN algorithm.

x_t The input provided by the data generation process $\mathcal{P}$ at time t.

y_t The classification output provided by the data generation process $\mathcal{P}$ at time t.

Privacy-Preserving

$\hat{x}, \hat{y}$ The encrypted values of x and y, respectively.

$\mathcal{D}$ In this domain, a CNN approximated to have only additions and multiplications.

$\mathcal{D}_\Theta$ In this domain, the encoded version of a CNN $\mathcal{D}$ with the parameters Θ.

Θ The set of encryption parameters (m, p, q).

$\tilde{\ell}$ The CNN $\mathcal{D}$ layer at which extract the features in transfer learning mode.

D The decryption function, requiring an encrypted value $\hat{y}$, the parameters Θ, and the secret key k_s.

E The encryption function, requiring an input x, the parameters Θ, and the public key k_p.

f In this domain, a CNN.

k_p, k_s A pair of public and secret keys, respectively.

m Polynomial modulus degree.

p Plaintext modulus.

q Ciphertext coefficient modulus.

x A generic input (image).

$x_\ell, \hat{x}_\ell$ The output of a CNN layer ℓ, either plain or encrypted.

y The classification of the input x.

Birdsong Detection Application

ψ The ToucaNet architecture (more specifically, its detector).

$\bar{x}_c$ In this domain, the number of columns of the spectrogram x

$\bar{x}_r$ In this domain, the number of rows of the spectrogram x.

$\hat{x}$ or $\bar{x}$ In this domain, a spectrogram computed from x.

d The dimension of the acquired audio (that might contains a bird call).

f_a The acquisition frequency of a given audio (that might contains a bird call).

h_l The distance between two consecutive windows of the Short Time Fourier Transform on a waveform x during the computation of the spectrogram.

n_{fft} The number of Fourier transforms used during the computation of the spectrogram through the Short Time Fourier Transform.

t_a The acquisition time of a given audio (that might contains a bird call).

x In this domain, x is the input waveform.

Solar Magnetogram Application

ϕ The longitude.

ϱ In this domain, the feature extractor operator is a (part of) either an AlexNet or a ResNet-101.

ς In this domain, the dimensionality reduction operator is the PCA.

ϑ The colatitude.

B_r Radial component of the magnetic field.

R_0 Sun Radius.

$s_{\vartheta,\phi}$ or S A synoptic map.

Other Used Symbols

$\varepsilon_{a,b}$ The percentage error between the two quantities a and b.

Part II

On-Device Deep Tiny Machine Learning

CHAPTER 5

Deep Tiny Machine Learning Solutions

The first approach to devise Deep Tiny Machine Learning solutions focuses on the design of inference-only solutions, whereas their training is carried out elsewhere, e.g., on the Cloud. In order to match the memory, computational, and energy requirements of the machine and deep learning solutions to the corresponding memory, computational, and energy technological constraints introduced by the IoT units, this chapter presents different approaches.

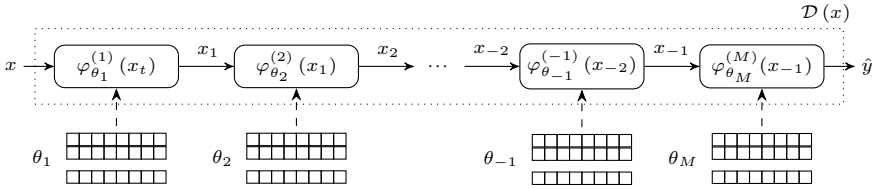
Section 5.1 deals with the reduction of DLM memory footprints, either by approximating the DLM pipeline or the precision of the DLMs' parameters.

Section 5.2 focuses on the computational load constraint by proposing Early-Exits within the DLM pipeline that can be optionally taken, skipping the remaining computation and thus reducing the mean computational load of the DLM.

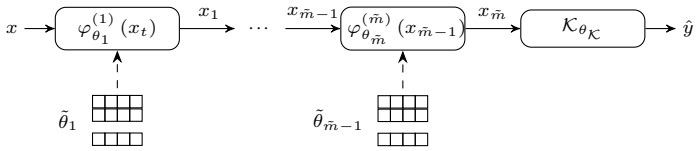
Finally, Section 5.3 presents an approach to make DLMs adaptive in presence of concept drift. Although this cannot be considered a pure TML approach (the training procedure during the adaptation is unfeasible for almost all the IoT units), it supports the triggering of DLM adaptation over time (in response to changes in the data generating process), a crucial ability for IoT units operating in real-world (possibly time-varying) environments (please refer to Chapter 6 for details).

5.1 Reducing Memory Complexity

Section 4.2 formalized the concept of Deep Learning Algorithm $\mathcal{D}$, its pipeline of layers, and the layer parameters θ_ℓ (with ℓ referring to the layer).



(a) *A generic DLM $\mathcal{D}$ architecture with M layers and 32-bit data parameters θ_ℓ, according to the definition in Eq.* (4.1)*, where a layer identified by a negative number $-x$ corresponds to the layer $M - x$.*



(b) *The approximated DLM $\hat{\mathcal{D}}$ architecture with $\tilde{m} \leq M$ layers after task dropping and the parameters $\tilde{\theta}_\ell$ after precision scaling with parameter $\tilde{q}$ from θ_ℓ, for each layer ℓ within $\mathcal{D}$. Optionally, a final classifier layer $\mathcal{K}$ with parameters $\theta_\mathcal{K}$ working on features $x_{\tilde{m}}$ at layer $\tilde{m}$ can be added.*

Figure 5.1: *A sketch of the proposed mechanisms to reduce the memory footprint of a Deep Learning Model $\mathcal{D}$.*

The memory footprint of DLMs algorithms can be reduced through the several solutions proposed in the related literature under the *approximation techniques* (see Section 3.2 for details). In particular, Section 5.1.1 details how to approximate the DLM pipeline using task dropping (e.g., layers removal) or pruning (e.g., reducing the number of filters within a convolutional layer) techniques, whereas Section 5.1.2 focuses on the precision scaling of the DLMs' parameters. Finally, Sections 5.1.3 proposes a methodology to design Deep Tiny Machine Learning models combining both the approaches, then evaluated in Section 9.1.

5.1.1 Task Dropping or Pruning

In the literature of approximating techniques (Mittal, 2016), the terms task dropping refers to all the solutions that reduce the computational load and the memory footprint by skipping the execution of some tasks associated with the processing chain. In our domain, task dropping refers to removing all the DLM layers that are not feasible with the considered IoT unit memory and computational constraints. Formally, instead of executing the whole DLM $\mathcal{D}$'s pipeline of M layers, only the first $\tilde{m} \leq M$ are kept and carried out on the IoT unit, defining the following approximate DLM model $\hat{\mathcal{D}}$:

$$\begin{aligned} \mathcal{D} &= \varphi^{(M)}_{\theta_M}(x_{M-1}) = \varphi^{(M)}_{\theta_M}\left(\varphi^{(M-1)}_{\theta_{M-1}}(x_{M-2})\right) = \varphi^{(M)}_{\theta_M}\left(\dots\left(\varphi^{(\tilde{m})}_{\theta_{\tilde{m}}}(x_{\tilde{m}-1})\right)\right), \\ \hat{\mathcal{D}} &= \varphi^{(\tilde{m})}_{\theta_{\tilde{m}}}(x_{\tilde{m}-1}). \end{aligned} \tag{5.1}$$

Other task dropping strategies can be envisioned as well, e.g., by considering non consecutive layers in the approximate model $\hat{\mathcal{D}}$. Similarly to task dropping, pruning techniques

can further reduce the memory footprint by approximating the single tasks. In this case, several solutions are specific to the considered DLM layer. For instance, pruning in convolutional layers might reduce the number of filters, whereas in fully-connected ones the number of neurons.

A classification layer $\mathcal{K}$ can be added at the end of the approximated DLM $\hat{\mathcal{D}}$ in order to design a complete approximated pipeline that can either address the original classification task or, more generally, a novel one. Formally,

$$\hat{\mathcal{D}} = \mathcal{K}_{\theta_{\mathcal{K}}} \left(\varphi^{(\tilde{m})}_{\theta_{\tilde{m}}} \left(x_{\tilde{m}-1} \right) \right), \tag{5.2}$$

where $\theta_{\mathcal{K}}$ represents the parameters of the classifier $\mathcal{K}$, that classifies on the activations $x_{\tilde{m}}$ extracted by the approximated model at its last layer $\tilde{m}$.

5.1.2 Precision Scaling

Precision scaling aims to change the precision (number of bits for the representation) of the inputs or intermediate operands to reduce the memory occupation (Mittal, 2016). In our specific case, precision scaling aims at reducing the memory occupation associated with the parameters θ_ℓ of $\mathcal{D}$ by considering approximated versions $\tilde{\theta}_\ell$ of θ_ℓ, for any layer ℓ within $\mathcal{D}$ or, more generally, for a subset of layers $\mathcal{L}$. The most common choice for parameters θ_ℓ is to rely on 32-bit floating-point data types. In this scenario, precision scaling approximates the parameters through rounding down to a fixed-point representation with 16, 8, or even 1 bit. In particular, in the latter case, namely binary parameters, the approximation can be done through the sign function on the original values, i.e., $\tilde{\theta}_\ell = sgn\left(\theta_\ell\right)$.

Formally, the resulting approximated model $\hat{\mathcal{D}}$ is:

$$\begin{aligned} \mathcal{D} &= \varphi^{(M)}_{\theta_M}\left(x_{M-1}\right) = \varphi^{(M)}_{\theta_M}\left(\varphi^{(M-1)}_{\theta_{M-1}}\left(x_{M-2}\right)\right) = \varphi^{(M)}_{\theta_M}\left(\ldots\left(\varphi^{(1)}_{\theta_1}\left(x\right)\right)\right), \\ \hat{\mathcal{D}} &= \varphi^{(M)}_{\tilde{\theta}_M}\left(x_{M-1}\right) = \varphi^{(M)}_{\tilde{\theta}_M}\left(\varphi^{(M-1)}_{\tilde{\theta}_{M-1}}\left(x_{M-2}\right)\right) = \varphi^{(M)}_{\tilde{\theta}_M}\left(\ldots\left(\varphi^{(1)}_{\tilde{\theta}_1}\left(x\right)\right)\right), \end{aligned} \tag{5.3}$$

where for each layer $\varphi^{(\ell)}_{\theta_\ell}$, for each ℓ in $\{1, \ldots, M\}$, the precision of its parameters θ_ℓ have been scaled to $\tilde{\theta}_\ell$. In particular, let $\tilde{q}$ be the parameter of the precision scaling modeling the number of bits in $\tilde{\theta}_\ell$ and its fixed-representation format. The most used fixed representation pair $\tilde{q}$ (with a few variants) employs three values: the total number of bits, the number of bits for the integer representation, and a value specifying the presence or not of a sign bit (the number of bits for the fractional part are derived from the other three values).

Interestingly, there is a parallelism between precision scaling and DL regularization. On the one hand, the rounding operation of precision scaling introduces a perturbation in the original weights θ_ℓs (Alippi, 2014). On the other hand, noise-addition or dropout mechanisms (Srivastava et al., 2014) are typically considered during the training of DLMs to make them more robust against perturbations, e.g., more tolerant to truncation/rounding operations. Examples of other techniques to achieve a similar level of robustness encompass the modification of the figure of merit optimized during learning (Srivastava et al., 2014).

Algorithm 5.1: The methodology computing the Pareto Front of the feasible Deep Learning Model approximations given the memory $\bar{m}$ and computational $\bar{c}$ constraints of the target IoT unit(s).

Input: DLM $\mathcal{D}$, Memory $\bar{m}$ and Computational $\bar{c}$ constraints, Classifier $\mathcal{K}$, Training set $\mathcal{T}$.
Parameters : The transformation function Ω from a DLM to its approximation.
Output: The Pareto Front of feasible solutions $\mathcal{F}$.

Initialize $\mathcal{F}_{all} \leftarrow \emptyset$. /* Feasible Solutions Set. */
/* Search of feasible approximations. */
foreach $\tilde{m} \in \{1, \ldots, M\}$ **do**
  foreach $\tilde{q} \in \{\tilde{q}_1, \ldots, \tilde{q}_{max}\}$ **do**
    /* Compute current approximation candidate. */
    Compute $\hat{\mathcal{D}} \leftarrow \Omega\left(\mathcal{D}, \tilde{m}, \tilde{q}, \mathcal{K}_{\theta_\mathcal{K}}\right) = \mathcal{K}_{\theta_\mathcal{K}}\left(\varphi^{(\tilde{m})}_{\tilde{\theta}_{\tilde{m}}}\left(x_{\tilde{m}-1}\right)\right)$.
    if $m_{\hat{\mathcal{D}}} \leq \bar{m}$ *and* $c_{\hat{\mathcal{D}}} \leq \bar{c}$ **then**
      $\mathcal{F}_{all} \leftarrow \mathcal{F}_{all} \cup \{\hat{\mathcal{D}}\}$. /* The candidate $\hat{\mathcal{D}}$ is feasible. */
/* Evaluate the feasible solutions. */
Initialize $\mathbb{T} \leftarrow \emptyset$
foreach $\hat{\mathcal{D}} \in \mathcal{F}_{all}$ **do**
  $\hat{\alpha} \leftarrow \text{train}\left(\hat{\mathcal{D}}, \mathcal{T}\right)$. /* Train $\hat{\mathcal{D}}$ with $\mathcal{T}$. */
  $\mathbb{T} \leftarrow \mathbb{T} \cup \langle \hat{\alpha}, m_{\hat{\mathcal{D}}}, c_{\hat{\mathcal{D}}} \rangle$.
$\mathcal{F} \leftarrow \text{pareto}\left(\mathcal{F}_{all}, \mathbb{T}\right)$. /* Compute the pareto front. */

5.1.3 Combine Task Dropping and Precision Scaling To Design Deep Tiny Machine Learning Models

Figure 5.1 shows the proposed mechanisms to reduce the memory footprint of a DLM $\mathcal{D}$ (Figure 5.1a) by creating its approximated version $\hat{\mathcal{D}}$ (Figure 5.1b) that is able to be executed on the IoT unit(s).

Formally, the methodology $\mathcal{M}$ to automatically design approximated Deep Tiny Machine Learning models is

$$\mathcal{F} = \mathcal{M}\left(\mathcal{D}, \Omega, \mathcal{T}, \bar{m}, \bar{c}\right), \tag{5.4}$$

where $\bar{m}$ and $\bar{c}$ are the memory capacity and the available computational of the considered IoT units, namely the introduced technological constraints, $\mathcal{T}$ is a training set, and Ω the transformation from the original DLM $\mathcal{D}$ to its approximation $\hat{\mathcal{D}}$, i.e.:

$$\hat{\mathcal{D}} = \Omega\left(\mathcal{D}, \tilde{m}, \tilde{q}, \mathcal{K}_{\theta_\mathcal{K}}\right) = \mathcal{K}_{\theta_\mathcal{K}}\left(\varphi^{(\tilde{m})}_{\tilde{\theta}_{\tilde{m}}}\left(x_{\tilde{m}-1}\right)\right), \tag{5.5}$$

where $\tilde{m}$ specifies how many layers to keep in the task dropping step, $\tilde{q}$ the parameters precision during their scaling, and the (optional) parameter $\mathcal{K}$ is the classifier to be added at the top of $\hat{\mathcal{D}}$.[7]

The result of the methodology $\mathcal{M}$ is the Pareto front $\mathcal{F}$ w.r.t. the training set $\mathcal{T}$ containing all the models $\hat{\mathcal{D}}$ that satisfy the constraints $\bar{m}$ and $\bar{c}$.

[7] To simplify the notation, the sets of possible values for $\tilde{m}$, $\tilde{q}$, and optionally $\mathcal{K}$, as well as the training set $\mathcal{T}$ are "included" within the input Ω of the methodology in Eq. (5.4).

Algorithm 5.1 shows the sketch of a possible implementation that encompasses two steps, which are detailed in the following.

Step 1: Identification of Feasible Solutions. This step aims at identifying all the feasible approximations $\hat{\mathcal{D}}_i$ of the original DLM model $\mathcal{D}$, i.e.,

$$\mathcal{F}_{all} = \left\{ \hat{\mathcal{D}} \text{ s.t. } m_{\hat{\mathcal{D}}} \leq \bar{m} \text{ and } c_{\hat{\mathcal{D}}} \leq \bar{c} \right\}, \tag{5.6}$$

generated via the transform Ω and the exploration of all possible $\tilde{m} = \{1, \ldots, M\}$ and $\tilde{q} = \{q_1, \ldots, q_{max}\}$ (Algorithm 5.1, Lines 2–6). In the most general configuration, during the exploration the methodology encompasses: a set of possible classifiers $\mathcal{K}$ since their parameters $\theta_{\mathcal{K}}$ impact the definition of feasible models; the possibility to define custom sets for the number of layers $\tilde{m}$ and the precision $\tilde{q}$ thus avoiding to check all the intermediate possible values; the possibility to consider mixed-precision mechanisms, i.e., where the precision $\tilde{q}$ vary across the layers of $\hat{\mathcal{D}}$. Obviously, the wide is the grid of possible solutions, the higher is the time the methodology requires to provide the results. Hence, a trade-off between the granularity of the exploration and its time span is expected.

Step 2: Identification of the Pareto Front. The second step takes as input the set $\mathcal{F}_{all}$ and provides as output the Pareto front $\mathcal{F}$ with respect to the training set $\mathcal{T}$ (Algorithm 5.1, Lines 7–11). More in detail, for each approximation $\hat{\mathcal{D}}_i \in \mathcal{F}_{all}$, the classifier $\mathcal{K}$ of $\hat{\mathcal{D}}_i$ is trained and evaluated on disjoint subsets of $\mathcal{T}$ producing a metric $\hat{\alpha}_i$.[8] The Pareto front $\mathcal{F}$ is then computed on the triples $\langle \hat{\alpha}_i, m_{\hat{\mathcal{D}}_i}, c_{\hat{\mathcal{D}}_i} \rangle$.

It is noteworthy to point out that the comparisons between two metrics $\hat{\alpha}_i$ and $\hat{\alpha}_j$ should take into account the confidence interval of the estimates, in particular when the number of samples is small. A simple way to do so within $\mathcal{M}$ is through a paired test to assess the classification comparison.

Selecting the Target Solution: a Discussion. The output of the methodology is a set of solutions. As a consequence, the choice of the DLM is left to the methodology users, e.g., an embedded system application designer. Such designer should select the solution within the Pareto front that is more suitable for the technological constraints of the application he is designing. To give an example, a solution with a smaller accuracy, memory, and time complexity might be preferable w.r.t. a more complex and accurate feasible solution in a real-time scenario, but the same reasoning no longer applies to a scenario where a higher accuracy is crucial disregarding the computational time.

A second aspect that might be critical for the proposed algorithm is the possibility to provide a solution that overfits the evaluation subset within $\mathcal{T}$, a common problem in the tools –namely autoML tools– that aims at finding the best machine or deep learning model for a given dataset (Fabris and Freitas, 2019; Wong et al., 2018). This potential issue has not been investigated in this work for the proposed methodology. However, such methodology considers technological constraints, hence potentially alleviating the overfitting issue since there is not the possibility to create an over-complex DLM. As a final remark, an alternative approach w.r.t. the proposed methodology is that of Neural

[8] Other approaches can be considered as well, e.g., Leave-One-Out, k-Fold Cross-Validation, and Bootstrap.

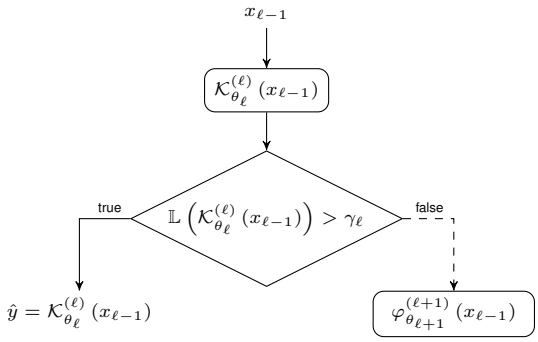


Figure 5.2: *The representation of a Gate-Classification layer $\varphi^{(\ell)}_{\theta_\ell}$. The output of the previous layer, i.e. $x_{\ell-1}$, is processed by the classifier $\mathcal{K}^{(\ell)}_{\theta_\ell}$. Thereafter, if the posterior probability $\mathbb{L}\left(\mathcal{K}^{(\ell)}_{\theta_\ell}(x_{\ell-1})\right)$ overcomes the threshold γ_ℓ, the classification $\mathcal{K}^{(\ell)}_{\theta_\ell}(x_{\ell-1})$ is final and provided as output (solid line), otherwise $x_{\ell-1}$ is forwarded to the subsequent layer $\varphi^{(\ell+1)}$ of the pipeline (dashed line).*

Architectural Search (Elsken et al., 2019; Li and Talwalkar, 2020), that explores different possible DL architectures in order to find the optimal one for the considered task or problem. Having said that, the drawbacks are the wider exploration space (that potentially grows indefinitely) and the need for training each explored candidate DL architecture, a strong limitation in all those scenarios where a few data are available.

5.2 Reducing Computational Complexity via Early–Exit: Gate-Classification

The term *Early–Exit* DLMs refers to all the models whose processing pipeline varies according to the input information content, i.e., there is the possibility to ends up the computation at intermediate layers, namely the *Early-Exit*s, hence skipping all the remaining computation. The introduction of *Early-Exit*s within Deep Tiny Machine Learning models does not affect the memory footprint of the algorithm (the introduction of such layers, instead, causes an increment of the footprint) but guarantees a mean reduction in the computational load of the model itself. This gain depends on how many times these Early-Exits are actually taken.

This section aims at addressing the problem by supporting the design of DLMs with one or more Early-Exits, whereas Section 9.2 provides the corresponding evaluation.

The Gate-Classification Layer. Figure 5.2 shows the proposed Early-Exit layer, called *Gate Classification* layer. The *Gate-Classification* layer is defined as follows:

$$\varphi^{(\ell)}_{\theta_\ell}(x_{\ell-1}) = \begin{cases} x_\ell = x_{\ell-1} & \text{if } \mathbb{L}\left(\mathcal{K}^{(\ell)}_{\theta_\ell}(x_{\ell-1})\right) < \gamma_\ell \\ \hat{y} = \mathcal{K}^{(\ell)}_{\theta_\ell}(x_{\ell-1}) & \textit{otherwise} \end{cases} \tag{5.7}$$

where $\mathcal{K}_{\theta_\ell}^{(\ell)}$ is a classifier that takes as input the activations $x_{\ell-1}$ of the $(\ell-1)$-th layer of the DLM and provides as output both a classification $\mathcal{K}_{\theta_\ell}^{(\ell)}(x_{\ell-1}) \in \Delta$ and the posterior probability $\mathbb{L}\left(\mathcal{K}_{\theta_\ell}^{(\ell)}(x_{\ell-1})\right) \in [0,1]$ of the predicted class. When the posterior probability is larger than an automatically computed threshold γ_ℓ, the classifier $\mathcal{K}$ has enough "confidence" to take a decision and the classification becomes the output of the DLM, i.e., $\hat{y} = \mathcal{K}_{\theta_\ell}^{(\ell)}(x_{\ell-1})$. In this case, all the layers from $\ell+1$ to M are skipped, hence saving computational load. Conversely, when the posterior probability is lower than γ_ℓ, the DLM proceeds to the next $\ell+1$ layer as in traditional DLMs, with $x_{\ell-1}$ being the forwarded activations.

Needless to say, $\mathcal{K}$ can be any deep learning-based (e.g., fully-connected layers) or even machine learning-based (e.g., k-Nearest Neighbors or Support Vector Machines) classifier with a posterior probability that can be (at least approximately) computed.

As a final remark, the proposed "gating" mechanism is close to the idea of having multiple paths within DLMs as in Deep Residual Learning (He et al., 2016) and Highway Networks (Srivastava et al., 2015; Zilly et al., 2017). Such paths allow information to flow across several layers of the DLMs. Our Gate-Classification mechanism extends this paradigm by introducing a control (i.e., the gate) on the information gained by the DLM in the hierarchical processing of the knowledge representation through the layers. These gates on the posterior probability of suitably trained classifiers (operating within the DLM) allow to regulate the processing of information through the DLM layers by either selecting a path directly connected to the output or by activating the next layer in the processing of the DLM.

Training Gate-Classification DLMs. The training of the Gate-Classification DLMs relies on two steps:

- learning the parameters θ_ℓs of the M DLM layers $\varphi_{\theta_\ell}^{(\ell)}$;
- learning the parameters γ of each Gate-Classification layer.

These two sequential steps are detailed in the following.

Training Gate-Classification DLMs: Learning the Parameters. During the first training step, the Gate-Classification layers are disabled, i.e., they forward the activations of the previous layer to the subsequent one in the DLM pipeline (this can be seen as having a value of $\gamma \geq 1$). In this way, the training of the DLM's parameters θ can be carried out with any of the available methods in the literature, e.g., gradient descend with back-propagation (Rumelhart et al., 1986). The core of this learning step is that all the classifiers, i.e., the gates and the final classifier, are jointly optimized.[9] Defined as $g < M$ the number of introduced Gate-Classification layers indices and as $\hat{\mathbf{y}}_\mathbf{i}$ the δ-dimensional posterior probability on the considered δ classes of Gate-Classification layer i, the loss function in the gradient-descend algorithm is the weighted sum of the classifiers' cross-entropy, i.e.,

$$\mathbb{L}\left(\hat{\mathbf{y}}_\mathbf{1}, \ldots, \hat{\mathbf{y}}_\mathbf{g}, \hat{\mathbf{y}}_\mathbf{M}, \mathbf{y}\right) = - w_1 \cdot \mathcal{H}\left(\hat{\mathbf{y}}, \hat{\mathbf{y}}_\mathbf{1}\right) - \ldots - w_g \cdot \mathcal{H}\left(\hat{\mathbf{y}}, \hat{\mathbf{y}}_\mathbf{g}\right) - w_M \cdot \mathcal{H}\left(\hat{\mathbf{y}}, \hat{\mathbf{y}}_\mathbf{M}\right) \quad (5.8)$$

[9] This approach is similar to that of the Inception CNN (Szegedy et al., 2015).

where $\hat{\mathbf{y}}$ represents the true posterior probability (i.e., 1 only on the correct classification class), $\mathcal{H}$ represents the cross-entropy function of two vectors, and the $g + 1$ weighting coefficients are hyper-parameters of the problem.

Training Gate-Classification DLMs: Learning the Thresholds γ. Once the training of the DLM's parameters has been carried out, the thresholds γ of the Gate-Classification layers can be computed. In particular, the value of γ should be the one that allows to activate the gate Early-Exit as much as possible (thus skipping the remaining pipeline) without reducing the global DLM classification capability.

Given a validation set $\mathcal{T}$, the value of γ is set as the minimum value $\hat{\gamma}$ such that there still exists an overlapping between the η-level confidence intervals of the empirical classification errors computed with $\hat{\gamma}$ and with $\gamma = 1$ on $\mathcal{T}$.

5.3 Adapting Deep Learning Models in Nonstationary Environments

Section 4.1 highlights that the data-generation process $\mathcal{P}$ might evolve over time (e.g., due to faults in the sensors, aging effects, seasonality). If the Deep Learning Model (Section 4.2) does not take into account this scenario, the effects on its performances might be catastrophic (Ditzler et al., 2015). This section aims to introduce an adaptive learning mechanism able to support the adaptation of DLMs in the presence of concept drift. In particular, the considered DLMs exploit a classification task and the concept drift within $\mathcal{P}$ are assumed to reflect on the classification metric (e.g., the accuracy) of the DLM itself. It is worth noting that the proposed approach is not specifically meant for TML since the suggested adaptation is not feasible on the IoT units. However, this work is a step into the direction of on-device learning, a problem addressed in Chapter 6.

Figure 5.3 and Algorithm 5.2 detail this proposed adaptive mechanism for DLMs in presence of concept drift (Disabato and Roveri, 2019). The adaptive DLM $\mathcal{D}$ comprises three further modules along with the DLM itself: a Change Detection Test ϑ, a Refinement Procedure, and the Adaptation procedure. More in details, an active approach based on a Change Detection Test ϑ continuously monitors the classification metric ϵ_t of $\mathcal{D}$ over time inspecting for variations (Algorithm 5.2–Lines 4–9). Once a change is detected at time $\hat{t}$, the Refinement Procedure estimates the time instant t_r the concept drift occurred (Algorithm 5.2–Line 10). The Adaptation procedure comprises two steps:

- the *Pipeline exploration* explores the DLM $\mathcal{D}$ pipeline to identify the layers that became *obsolete* due to the concept drift, hence having to be adapted to the new working conditions (Algorithm 5.2–Lines 11– 13);
- the *Retraining phase* re-trains only the *obsolete* layers in a transfer learning fashion (Yosinski et al., 2014) by relying on the input subsequence corresponding to the novel samples after the concept drift, i.e., all the data after the estimated change time t_r (Algorithm 5.2–Line 14).

The DLM has been already formalized in Section 4.2, whereas the remaining modules are defined in the following. Finally, Section 9.3 experimentally evaluates this adaptive DLM.

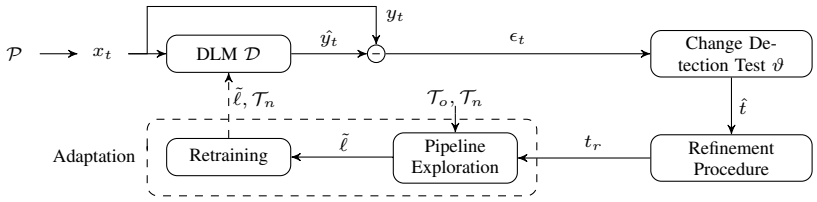


Figure 5.3: *The adaptive mechanism for Deep Learning Models in presence of concept drift. This mechanism encompasses four modules: the DLM $\mathcal{D}$ classifying the inputs, the Change Detection Test ϑ monitoring the DLM errors ϵ_t over time t inspecting for changes, the Refinement Procedure estimating the time t_r the concept drift occurred, and the Adaptation module adapting the DLM to the concept drift.*

5.3.1 Change-Detection Test and Refinement Procedure

Algorithm 5.2 presents the CDT in the most general way, i.e., at the current time instant t it computes the metric s_t is working on (Algorithm 5.2–Line 7). After that, the CDT applies its decision function ϑ either on the computed metric s_t or on all the previously computed metrics $s_1, \ldots, s_t$ from time instant 1 to t (Algorithm 5.2–Line 8). The CDT detects a change in the data generation process $\mathcal{P}$ if the output g_t of ϑ at time instant t overcomes a user-defined threshold h (Algorithm 5.2–Line 9).

This Section, instead, details the Change-Detection Test along with the Refinement Procedure in the case of a classification problem. However, similar solutions can be devised for other problems, e.g., regression or multi-class classification.

Under the test-then-train formalization described in Section 4.1, once $\mathcal{D}$ produced the classification $\hat{y}_t$ of the input x, the adaptive mechanism can compute the classification error ϵ_t, that is defined as

$$\epsilon_t = \begin{cases} 1 & \hat{y}_t \neq y_t, \\ 0 & otherwise. \end{cases} \tag{5.9}$$

It is crucial to stress that changes in the data-generation process $\mathcal{P}$ are assumed to reflect on the classification error ϵ_t in order to be detected by this adaptive mechanism. All the changes not affecting the classification error can be neglected and considered as irrelevant since they do not impact the $\mathcal{D}$ performances. For this reason, the classification error ϵ_ts is monitored over time by means of a CDT ϑ aiming at detecting variations in its probability distribution.

Among the solutions present in the literature, the considered CDT ϑ is the non parametric CUSUM test suggested by (Lorden et al., 1971). Such a CDT represents a non-parametric version of the well-known Cumulative Sum (CUSUM) test (Basseville et al., 1993; Page, 1954), suitably modified to operate on Bernoulli distributions, i.e., the outcomes of the classification error ϵ_t at each time instant t.[10] The reason to chose the CUSUM CDT is twofold. First, the parametric CUSUM (Page, 1954) provides theoretically guarantees, such as the possibility to define the average run length (i.e., the mean number of processed inputs before an action is taken), to compute bounds on detection delay (i.e., the time between the change and its detection) and the distance between two false

[10] Please refer to Section 6.3.2 for a definition of such CUSUM test in similar settings.

Algorithm 5.2: The Adaptive Deep Learning Model for Concept Drift. The DLM $\mathcal{D}$ enriched with adaptative functionalities is considered as an input.

Input: DLM $\mathcal{D}$, CDT ϑ.
Parameters : History Window Size ϖ, CDT threshold h, Adaptation inputs Ξ, Pipeline Exploration Target Layers $\mathcal{L}$, Paired t-test Confidence α.

Initialize $W \leftarrow \emptyset$. /* History Window. */
/* Loop over samples arriving at time t. */
foreach $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
  Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$.
  $W \leftarrow W \cup \{(x_t, y_t)\}$. /* Update History Window. */
  if $|W| \geq \varpi$ **then**
    $W \leftarrow W \setminus \{(x_{t-\varpi}, y_{t-\varpi})\}$.
  Compute CDT metric s_t. /* Active Step. */
  Apply CDT $g_t \leftarrow \vartheta(s_1, \ldots, s_t)$.
  if $g_t \geq h$ **then** /* Change Detection Check. */
    Estimate Real Change Time t_r. /* Refinement Procedure. */
    $\mathcal{T}_o \leftarrow \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \in [t_r - \Xi, t_r)\}$. /* Old Samples. */
    $\mathcal{T}_n \leftarrow \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \in [t_r, t_r + \Xi)\}$. /* Novel Samples. */
    $\tilde{\ell} \leftarrow p_e(\mathcal{D}, \mathcal{T}_o, \mathcal{T}_n, \mathcal{L}, \alpha)$ /* Pipeline Exploration. */
    Update $\mathcal{D}$ with $\mathcal{T}_n$ from layer $\tilde{\ell}$. /* Retraining. */

alarms, and the asymptotic or first-order optimality to the change detection problem (Lorden et al., 1971; Basseville et al., 1993; Pollak, 1987). Second, the non-parametric variant has the advantage of providing an estimate t_r of the time instant each detected change occurred, hence it acts also as a Refinement Procedure. Other solutions for RP could be found in (Alippi et al., 2017), e.g., based on statistical hypothesis tests or change-point methods.

Independently from the adopted CDT ϑ (the suggested one or another), they usually suffer from false positive and negative detections (Alippi et al., 2017). In the considered scenario, a false positive detection would induce an unnecessary adaptation (possibly increasing the computational load of the adaptive machine learning system). In contrast, a false negative detection could affect the classification accuracy since the DLM $\mathcal{D}$ operates in an obsolete configuration.

5.3.2 Adaptation: Pipeline Exploration and Retraining

When the CDT detects a change, whose RP's estimated change time is t_r, the adaptation stage begins, with the goal to adapt the DLM $\mathcal{D}$ to concept drift.

The trivial way to adapt $\mathcal{D}$ is to retrain it with all the novel content, i.e., all the sequence $\{(x_{t_r}, y_{t_r}), \ldots, (x_t, y_t)\}$ following the (estimated) change time. Unfortunately, this approach might lead to unsatisfactory results since the number of available inputs during the retraining is generally much lower than those typically considered in DLM training. Moreover, the time required to train the whole processing chain of $\mathcal{D}$ could not be compatible with the need to adapt it to concept drift quickly.

Differently, Figure 5.4 shows the proposed adaptation module that relies on the fact

that the processing chain of $\mathcal{D}$ is organized into layers characterized by an increasing granularity of the representation. Following the "transfer learning" approach (Yosinski et al., 2014), the goal is to transfer knowledge, i.e., part of the processing chain, from the DLM operating before the concept drift detection to the one operating after. To achieve this goal, the proposed adaptation module comprises two steps. The *Pipeline exploration* (Figure 5.4b) scans the processing chain of $\mathcal{D}$ to identify the layer $\tilde{\ell}$ after which all the remaining layers have to be retrained because obsolete. The *Retraining* (Figure 5.4c) retrains the layers from $\tilde{\ell}$ to M of $\mathcal{D}$. As a consequence, all the layers before $\tilde{\ell}$ are transferred from the DLM operating before the concept drift to the one operating after.

It is assumed that the structure of the processing chain of $\mathcal{D}$ is left unchanged during this adaptation procedure, i.e., no layer is added or removed. This assumption could be weakened by considering an exploration technique to identify the best architecture of the processing chain during the adaptation phase provided that enough (time and) data $\{(x_{t_r}, y_{t_r}), \ldots, (x_t, y_t)\}$ are available.

Pipeline exploration. Let $\mathcal{T}_o$ and $\mathcal{T}_n$ be two Ξ-dimensional batches of tuples acquired before and after the estimated change time t_r, respectively, defined as follow:

$$\mathcal{T}_o = \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \in [t_r - \Xi, t_r)\}, \tag{5.10}$$

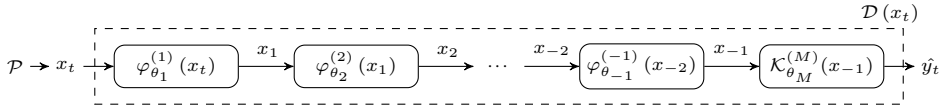
$$\mathcal{T}_n = \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \in [t_r, t_r + \Xi)\}, \tag{5.11}$$

where Ξ corresponds to the number of inputs acquired between t_r and the current time instant t. Obviously, it is possible to define a minimum value for Ξ to guarantee that at least a given number of inputs is available for the adaptation procedure. This comes at the expense of a possible delay in the activation of the procedure itself. Moreover, the algorithm is assumed to keep the last ϖ samples as history window (Algorithm 5.2– Lines 4– 6).

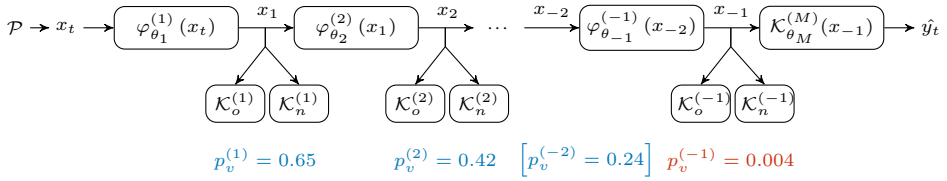
Let $\mathcal{L} \subseteq \{1, \ldots, M\}$ be the set of "convenient" layers, the pipeline exploration has to explore in $\mathcal{D}$, e.g., fully-connected, sub-sampling, and non-linearity layers. Figure 5.4 shows the pipeline exploration, whose goal is to find the layer $\tilde{\ell} \in \mathcal{L}$ after which the layers in $\mathcal{D}$ became obsolete (due to the concept drift).

To achieve this goal, all the layers in $\mathcal{L}$ are evaluated to check if they provide the same classification ability on $\mathcal{T}_o$ and $\mathcal{T}_n$, looking for the first layer $\tilde{\ell} \in \mathcal{L}$ at which this hypothesis does not hold. More specifically, for each layer $\ell \in \mathcal{L}$, two classifiers $\mathcal{K}_o^{(\ell)}$ and $\mathcal{K}_n^{(\ell)}$ are trained on (a training subset of) $\mathcal{T}_o$ and $\mathcal{T}_n$, respectively. Both classifiers belong to the same family of classifiers (e.g., Neural Networks, Support Vector Machines, or Decision Trees), share the same architecture (e.g., the number of layers and neurons in Neural Networks), and are trained in the same conditions (e.g., the same initialization procedure with a shared random seed).

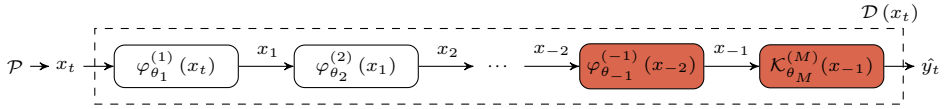
Let $E_o^{(\ell)}$ and $E_n^{(\ell)}$, for each $\ell \in \mathcal{L}$, be the sequence of binary classification errors (i.e., 1 for a misclassified sample, 0 otherwise) on (a test subset of) $\mathcal{T}_o$ and $\mathcal{T}_n$, respectively. To assess that both the classifiers $\mathcal{K}_o^{(\ell)}$ and $\mathcal{K}_n^{(\ell)}$ provides the same classification capabilities, a *Paired t-test* statistical hypothesis test (Lehmann and Romano, 2006) is employed aiming at making inference about the difference in the mean of the sets $E_o^{(\ell)}$ and $E_n^{(\ell)}$. The null hypothesis of the test equals to zero the difference between the means of the two sets, whereas the alternative hypothesis assumes such difference is not equal to zero.



(a) *A generic DLM $\mathcal{D}$ architecture with M layers, according to the definition in Eq.* (4.1)*, where $\mathcal{K}^{(M)}$ is here renamed to identify the last layer as a classification one and where a layer identified by a negative number $-x$ corresponds to the layer $M - x$.*



(b) *The pipeline exploration of the DML model in Figure 5.4a, with $\mathcal{L}$ encompassing all layers. For each layer ℓ in $\mathcal{L}$, a pair of classifiers $\mathcal{K}^{(\ell)}_o$ and $\mathcal{K}^{(\ell)}_n$ is trained on $\mathcal{T}_o$ and $\mathcal{T}_n$, respectively (see Eqs.* (5.10)– (5.11)*). The first layer $\tilde{\ell}$ for which the p-value of the Paired t-test $p^{(\tilde{\ell})}_v$ is lower than the confidence α is in this example $\tilde{\ell} = M - 1$, with $\alpha = 0.05$.*



(c) *The adapted DML $\mathcal{D}$, after the retraining phase. In particular, the layers involved in this phase, i.e., $\varphi^{(-1)}_{\theta_{-1}}(x_{-2})$ and $\mathcal{K}^{(M)}_{\theta_M}(x_{-1})$, are highlighted, whereas all the other remaining layers are left unchanged (their parameters θ_ℓs are the same of Figure 5.4a).*

Figure 5.4: *A detail of the adaptation procedure of the proposed adaptive mechanisms for learning deep learning models in presence of concept drift.*

Let

$$p^{\ell}_v = \textit{paired t-test}\left(E^{(\ell)}_o, E^{(\ell)}_n, \alpha\right), \tag{5.12}$$

be the p-value of the *Paired t-test* statistical hypothesis test applied to layer ℓ, for each $\ell \in \mathcal{L}$, with α the confidence of the hypothesis test. It is noteworthy to point out that having multiple hypothesis tests, corrections on the confidence α could be considered (e.g., the Bonferroni correction (Bonferroni, 1936; Dunn, 1961)).

The first obsolete layer $\tilde{\ell}$ is then

$$\tilde{\ell} = \min_{\ell \in \mathcal{L}} \left\{ \ell \mid p^{\ell}_v < \alpha \right\}, \tag{5.13}$$

i.e., $\tilde{\ell}$ is the first layer of $\mathcal{D}$ for which the *Paired t-test* rejects the null hypothesis with confidence $1 - \alpha$, where the ordering of layers reflects that of the DML $\mathcal{D}$ itself. In the example given in Figure 5.4b, $\tilde{\ell}$ is $M - 1$.

Retraining. The *retraining* stage is activated by the pipeline exploration in order to learn, for each layer $\ell = \tilde{\ell}, \ldots, M$ of the DLM $\mathcal{D}$, its parameters θ_ℓ. This task depends on the type of employed DLM, e.g., for CNNs, a learning algorithm based on gradient descent (e.g., (Glorot and Bengio, 2010; Krizhevsky et al., 2012)) on the whole novel knowledge

set $\mathcal{T}_n$, with a number of epochs balancing the achieved accuracy after adaptation and the training time. In the example given in Figure 5.4, the retraining stage employs the layers $M-1$ and M, whose resulting adapted DML $\mathcal{D}$ is in Figure 5.4c.

As a final remark, if the cardinality of $\mathcal{T}_n$ is too small for the retraining stage, such stage can be postponed until enough novel knowledge is acquired and, in the meanwhile, the classifier $\mathcal{K}_n^{(M)}$ provides the output for the DML $\mathcal{D}$.

5.3.3 Discussion

There are two critical aspects of the proposed adaptation mechanism. The first one is the magnitude of change. Small changes might have a limited but not negligible impact on the DLM accuracy. As a consequence, a trade-off is expected between the need to detect these changes and to shrink towards zero the number of false-positive detections (i.e., detecting a change that has not occurred). This trade-off can be modeled with a further hyper-parameter that defines the minimum change magnitude in terms of accuracy impact, i.e., the minimum accuracy drop that can be led back to a change.

In the second order, the adaptation parameters have to be set carefully to trade off the need for either a fast or accurate adaptation. This choice reflects on the number of layers the pipeline exploration method considers, but in particular on the minimum number of samples Ξ acquired before starting the pipeline exploration itself (the more the samples, the higher the confidence in defining the set of obsolete layers at the expense of a higher exploration time) or the retraining. In the latter case, a small number of samples may result in an over-fitted model, whereas a too large number of samples is prone to be unfeasible in time-constrained application scenarios.

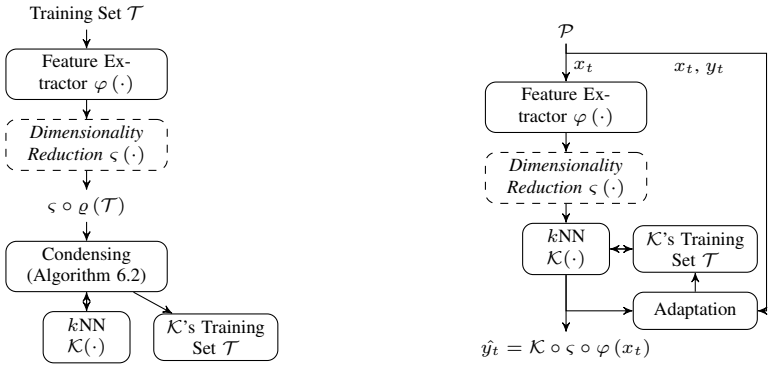
CHAPTER 6

On-Device Tiny Machine Learning

Chapter 5 presented various ways to design Tiny Machine Learning (TML) solutions encompassing Deep Learning Models (DLM), namely, Deep Tiny Machine Learning solutions. Such approaches focus on approximation, pruning, and quantization mechanisms to reduce the memory and computational demand of DLMs. Although these solutions run on embedded systems and IoT units, their training is typically carried out on high-performing units (such as Cloud or Edge-Computing systems), with very few papers in the related literature proposing on-device incremental learning mechanisms (Cai et al., 2020a; Disabato and Roveri, 2020; 2021).

The ability to learn TML models directly on the devices is crucial to improve the accuracy over time by exploiting fresh information coming from the field, and to deal with concept drift, i.e., variations in the statistical behavior of the data generating process, a quite common situation in real-world applications (e.g., due to seasonality or periodicity effects, faults affecting sensors or actuators, changes in the user's behavior, or aging consequences). Failing to adapt TML models to concept drift results in a (possibly dramatic) decrease in the accuracy over time (Ditzler et al., 2015).

This chapter aims at addressing this challenge by introducing, for the first time in the literature, a Tiny Machine Learning algorithm for Concept Drift (TML-CD) that can learn directly on the IoT unit and adapt the knowledge base in response to a concept drift (thus tracking the evolution of the data generating process). Differently from the solution presented in Section 5.3, this approach is fully on-device, i.e., both the inference and the adaptation steps are executed on the considered IoT unit. To achieve this goal, the papers (Disabato and Roveri, 2020; 2021) introduced four different adaptation mechanisms



(a) *Configuration: condensing of training set $\mathcal{T}$.* **(b)** *Testing or Operational Life.*

Figure 6.1: *The proposed architecture of the proposed solution for Tiny Machine Learning for Concept Drift (TML-CD) $\mathcal{D}$ on embedded systems and IoT units.*

(i.e., incremental, passive, active, and hybrid), each of which has its advantages, issues, effectiveness, and behavior. The hybrid solution is the suggested approach among these mechanisms thanks to its ability to trade-off adaptation with memory demand.

Section 6.1 presents the deep TML-CD architecture, that encompasses two main operating stages: the configuration (Section 6.2) and the testing (Section 6.3). In particular, being the adaptation within the testing stage, Section 6.3 presents also the four proposed adaptation mechanisms.

6.1 On-Device Deep Tiny Machine Learning for Concept Drift

Section 4.1 formalized the data-generation process $\mathcal{P}$ in nonstationary conditions along with the considered classification problem with δ classes. $\mathcal{P}$ provides at each time instant t a pair (x_t, y_t) where x_t is the input and $y_t \in \Delta$ the corresponding label. Moreover, Section 4.2 formalized the concept of Deep Learning Model and of the operators used to define the proposed solution.

Figure 6.1 shows the general architecture of the proposed solution for Tiny Machine Learning for Concept Drift (TML-CD) $\mathcal{D}$, which comprises the following five different modules:

- **Feature Extractor ϱ.** The *Feature Extractor* extracts features from the input x_t. As in (Alippi et al., 2018), the Feature Extractor is a pre-trained DLM approximated by means of Task-Dropping (e.g., pruning of layers), Precision Scaling (e.g., weights precision reduction), or both, to satisfy the constraints on computation, memory, and energy characterizing the IoT units running $\mathcal{D}$ (see Section 5.1 for details).

- **Dimensionality Reduction Operator ς** The *Dimensionality Reduction Operator* ς (that can be optionally activated) reduces the dimensionality of features extracted by ϱ. In this Chapter, among the approaches presented in (Disabato and Roveri, 2020), the *Filter-Selection without supervised information* is adopted. This technique selects the f out of F filters of the last ϱ convolutional layer (and its subse-

Algorithm 6.1: A sketch of the proposed on-device deep tiny machine learning solution for concept drift.

Input: Training Set $\mathcal{T}$, Feature Extractor $\varsigma \circ \varrho$.
Parameters : Number of neighbors k.
1 Preprocess $\mathcal{T}$. /* Initialization. */
2 Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho\,(\mathcal{T})$.
3 Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
/* Loop over samples arriving at time t. */
4 **foreach** $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
5 Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$.
6 *Adaptation of* $\mathcal{D}$.

quent batch-normalization channels, if any) providing the highest mean activation on publicly available benchmarks or datasets.

It is crucial to point out that the adaptation step of $\mathcal{D}$ does not affect the feature extractor ϱ nor the dimensionality reduction operator ς, which are therefore fixed over time. Moreover, since the choice of the f filters to keep does not rely on the specific data-generation process $\mathcal{P}$, the block $\varsigma \circ \varrho$ can be defined at design time and before the porting of $\mathcal{D}$ on the IoT units. This is the reason why $\varsigma \circ \varrho$ is an input to our algorithm, and ς takes part in the choice of the approximated DL-feature extractor.

- ***k*NN Classifier** and **Training Set $\mathcal{T}$**. The kNN (Altman, 1992) *classifier* $\mathcal{K}(\cdot)$, whose input is either the output of the dimensionality-reduction operator $\varsigma \circ \varrho$ or that of the feature extractor ϱ (when no dimensionality reduction is considered), provides the classification $\hat{y}_t$ of the input x_t, while $\mathcal{T}$ is its training set. From the algorithmic point of view, the kNN is a statistical classifier based on majority voting, i.e., the predicted class corresponds to the majority class of the k nearest neighbors of the input sample within $\mathcal{K}$'s training set $\mathcal{T}$. Interestingly, it does not require a training phase, but only the initialization of its training set $\mathcal{T}$. Unless otherwise specified, the parameter k, i.e., the number of neighbors, is set to the ceiling of the square root of the available samples, as suggested in (Alippi et al., 2013).

- **Adaptation Module**. The adaptation module receives as input the sample x_t and its kNN $\mathcal{K}$ prediction $\hat{y}_t$ and, when the supervised information y_t is available, it updates the TML-CD solution $\mathcal{D}$ so as to make it adaptive over time to concept drift. Among the four presented modules, the adaptation involves only the $\mathcal{K}$'s training set $\mathcal{T}$ (Losing et al., 2016; Roseberry and Cano, 2018). The kNN classifier adaption indeed requires to simply add the new supervised information (x_t, y_t) to its training set $\mathcal{T}$.

Algorithm 6.1 details how the proposed TML-CD $\mathcal{D}$ works. More in detail, the TML-CD $\mathcal{D}$, which receives in input a feature extractor along with a dimensionality reduction operator ($\varsigma \circ \varrho$) and an initial training set $\mathcal{T}$, comprises two different stages: configuration and testing.

The *configuration* stage, detailed in lines 1–3 and shown in Figure 6.1a, encompasses an initial *preprocessing step* where the training set $\mathcal{T}$ is preprocessed to reduce the memory occupation (line 1) by means of a condensing mechanism (Algorithm 6.2). Once the

Algorithm 6.2: The Condensed Nearest Neighbor (Hart, 1968).

Input: Training Set $\mathcal{T}$.
Output: Condensed Representation $\bar{\mathcal{T}} \subseteq \mathcal{T}$.

```
                    /* H contains one sample, D all the others.  */
```

1 Initialize $H \leftarrow \{t \in \mathcal{T}\}$ and $D \leftarrow \mathcal{T} \setminus H$.
2 Initialize the kNN $\mathcal{K}$ with H.
3 **do**
4 **foreach** $t \leftarrow (x, y) \in D$ **do**
5 Predict $\hat{y} \leftarrow \mathcal{K}(t)$.
6 **if** $\hat{y} \neq y$ **then** /* Condensing Update: */
7 $D \leftarrow D \setminus \{t\}$. /* Move t from D to H. */
8 $H \leftarrow H \cup \{t\}$.
9 (Re–)train the kNN $\mathcal{K}$ with H.
10 **while** *H and D are modified in the* foreach *loop.*
11 **return** $\bar{\mathcal{T}} \leftarrow H$

preprocessing step has been carried out, the knowledge base of the kNN classifier is initialized on the features extracted from the preprocessed training set $\mathcal{T}$, i.e., the training set of $\mathcal{K}$ is $\varsigma \circ \varrho\,(\mathcal{T})$. Section 6.2 will detail the configuration stage.

After the completion of the configuration stage, the TML-CD solution $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$ enters the *testing* stage where it is able to operate on the novel incoming samples provided by the data-generating process $\mathcal{P}$ (lines 4–6). At each time instant $t = 1, 2, \ldots$, the proposed solution $\mathcal{D}$ receives in input x_t and provides the output $\hat{y}_t = \mathcal{D}(x_t)$ (line 5). Then, when the supervised information y_t about x_t is made available as per the "test-and-train" approach, it activates the adaptation step (line 6). Section 6.3 will detail the testing stage by describing the proposed adaptive mechanisms for TML-CD.

6.2 The Configuration Stage: Condensing the Training Set $\mathcal{T}$

The kNN classifier has the great advantage of not requiring a proper training phase. However, this advantage comes, in principle, at the expense of the following two drawbacks. First, a kNN-based classifier requires to store all the data of the training set. Second, the more significant the amount of the training data, the higher the time to provide a classification in output. These drawbacks are more severe as the samples within $\mathcal{T}$ increase.

The related literature addresses these two issues from three different perspectives.

First, condensing techniques (Hart, 1968; Tomek, 1976b) aim at identifying the smallest subset of training data that can correctly classify all the training samples. Second, editing techniques (Laurikkala, 2001; Tomek, 1976a; Wilson, 1972) instead reduce the number of stored samples by removing the noisy ones, i.e., those not agreeing with their neighborhoods. Third, (Smith et al., 2014) proposed to train a supervised parametric classifier on available data and to remove all the samples having a classification probability below a hard threshold.

In this work, we focus on the first approach and, in particular, on the Condensed Nearest Neighbour algorithm. In more detail, $\mathcal{D}$ applies this algorithm during the preprocessing step (Algorithm 6.1–Line 1) in order to optimize both the memory and computational

Algorithm 6.3: The Condensing-in-Time (Passive).

Input: Training Set $\mathcal{T}$, Feature Extractor $\varsigma \circ \varrho$.
Parameters : Maximum number of training samples p.

Compute $\mathcal{T} \leftarrow \bar{\mathcal{T}}$ with Algorithm 6.2. /* Condense $\mathcal{T}$. */
Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho(\mathcal{T})$.
Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
/* Loop over samples arriving at time t. */
foreach $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
  Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$
  if $\hat{y}_t \neq y_t$ **then** /* Passive Update. */
    $\mathcal{T} \leftarrow \mathcal{T} \cup (x_t, y_t)$
    if $|\mathcal{T}| > p$ **then** /* Window Size Check. */
      $(x_{\tilde{t}}, y_{\tilde{t}}) \leftarrow \arg\min_{\bar{t}} \{(x_{\bar{t}}, y_{\bar{t}}) \in \mathcal{T}\}$
      $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(x_{\tilde{t}}, y_{\tilde{t}})\}$
    Update $\mathcal{D}$ with $\mathcal{T}$

requirements of the classifier $\mathcal{K}$. More specifically, given a N-dimensional training set $\mathcal{T} = \{(x_t, y_t), t = 1, \ldots, N\}$, the preprocessing step computes the condensed representation of T, i.e., the minimum subset $\bar{\mathcal{T}} \subseteq \mathcal{T}$ for which $\mathcal{K}$ is able to correctly classify all the samples in $\mathcal{T}$. Algorithm 6.2 shows the pseudo-code of the condensing algorithm proposed by (Hart, 1968) that is employed in the preprocessing step. Section 9.4.4 experimentally evaluates the impact on accuracy and memory demand of this condensing stage, highlighting that the significant savings in terms of memory come at the expense of a negligible drop in accuracy (in stationary conditions).

6.3 The Testing Stage: Adapting $\mathcal{T}$

The adaptation module, which is the core of the proposed $\mathcal{D}$, has been declined from four different perspectives, differing in the type of adaptation mechanism therein employed:

- **Incremental Update**. This baseline approach is particularly useful when a very few data are available during the configuration stage. The adaptation simply adds (all) the incoming samples to its knowledge base, without any explicit change detection nor a passive mechanism to select the samples that have to be stored;

- **Passive Update** (Section 6.3.1). The adaptation module relies on a fully passive approach where the adaptation is carried out at each new incoming supervised samples, without requiring an explicit detection of a change in the data-generating process $\mathcal{P}$;

- **Active Update** (Section 6.3.2). This adaptation module relies on a CDT to detects changes in $\mathcal{P}$. Once a change is detected, the algorithm adapts $\mathcal{K}$ accordingly;

- **Hybrid Update** (Section 6.3.3). The hybrid adaptation module integrates the passive approach with a CDT to speed up the adaptation stage exactly when needed.

6.3.1 Passive Update: the Condensing-in-Time approach

The passive approach, called *Condensing-in-Time* (CIT) algorithm, updates the training set $\mathcal{T}$ every time a new supervised sample is available. Algorithm 6.3 presents the CIT algorithm.

It receives in input the feature extractor along with a dimensionality reduction operator $\varsigma \circ \varrho$ and a training set $\mathcal{T}$, whose condensed representation $\bar{\mathcal{T}} \subseteq \mathcal{T}$ (see Algorithm 6.2) is used to initialize the training set $\mathcal{T}$ of the kNN. Once initialized, the CIT–$\mathcal{D}$, i.e., the $\mathcal{D}$ solution implementing the CIT adaption module, is ready to classify novel incoming samples. The CIT passively updates the $\mathcal{K}$'s knowledge set $\mathcal{T}$ at every time instant t for which the supervised information, i.e., the true label y_t, is available. More in details, CIT–$\mathcal{D}$ adds the sample x_t and its true label y_t (at time instant t) to the kNN $\mathcal{K}$ knowledge set $\mathcal{T}$ if and only if x_t is misclassified, i.e., $\hat{y}_t \neq y_t$ (Algorithm 6.3, Lines 6–11). This idea is inspired by the condensing algorithm update, shown at Lines 6–9 in Algorithm 6.2, but it is here tailored to the time evolution of the data-generating process.

It is worth noting that the CIT algorithm can only add a new supervised sample to the knowledge set $\mathcal{T}$ of the kNN $\mathcal{K}$, hence potentially introducing critical issues in the memory and computational demand of the kNN when the number of samples in $\mathcal{T}$ increases.

The CIT algorithm employs two different solutions in order to keep under control the cardinality of $\mathcal{T}$.

The former introduces a maximum number of samples p that can be stored, i.e., $|\mathcal{T}| \leq p$, being $|\cdot|$ cardinality operator. Hence, every time the adaptation stage introduces a sample in $\mathcal{T}$ overcoming this limit, the oldest sample is removed (Algorithm 6.3, Lines 8–10). As a consequence, the solution $\mathcal{D}$ based on the CIT classifier operates on the last p supervised samples introduced in $\mathcal{T}$. Besides, this mechanism allows also to remove old samples in $\mathcal{T}$ by introducing only misclassified samples, i.e., those bringing more information to the classifier.

The latter relies on a probability p_a to decide whether to ignore an error or to add it to the training set $\mathcal{T}$. The idea is the following: under stationary conditions, the TML-CD $\mathcal{D}$ has a constant accuracy υ_0 over time, and the rate of samples added to the training set is $\bar{\upsilon_0} = 1 - \upsilon_0$ (that corresponds to the error rate). As a consequence, the cardinality of $\mathcal{T}$ continuously increases without affecting the accuracy υ_0, a side effect of the CIT algorithm. Ideally, p_a should be close to zero (i.e., $\mathcal{T}$ cannot change) in stationary conditions and close to one (i.e., all the errors are added to $\mathcal{T}$) when there is a change affecting the accuracy.

Although many solutions can be developed, the following simple solution is advised. The CIT algorithm keeps trace of the mean value of υ_0 over time, e.g., by computing the exponential mean $\tilde{\upsilon_0}$, and computes its first derivative $\tilde{\upsilon_0'}$. Since in stationary conditions $\tilde{\upsilon_0}$ is assumed to remain roughly constant, the value of $\tilde{\upsilon_0'}$ is close to zero. The probability p_a is then a function that receives as input $\tilde{\upsilon_0'}$ and returns a value close to 0 when $\tilde{\upsilon_0'} \to 0$ and close to 1 as $\tilde{\upsilon_0'}$ goes far from 0, such as the absolute value of hyperbolic tangent or the hyperbolic secant of $\tilde{\upsilon_0'}$ reciprocal:

$$p_a = \left| tanh\left(\iota \cdot \tilde{\upsilon_0'}\right) \right| \qquad \text{or} \qquad p_a = sech\left(\frac{1}{\iota \cdot \tilde{\upsilon_0'}}\right), \tag{6.1}$$

where ι is an additional parameter defining the slope of the probability p_a curve.

Algorithm 6.4: The Active Tiny kNN.

Input: Feature Extractor $\varsigma \circ \varrho$, CDT ϑ, Training Set $\mathcal{T}$.
Parameters : History Window Size ϖ, CDT threshold h.

1 Compute $\mathcal{T} \leftarrow \bar{\mathcal{T}}$ with Algorithm 6.2. /* Condense $\mathcal{T}$. */
2 Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho(\mathcal{T})$.
3 Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
4 Initialize $W \leftarrow \emptyset$. /* History Window. */
/* Loop over samples arriving at time t. */
5 **foreach** $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
6 Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$.
7 $W \leftarrow W \cup \{(x_t, y_t)\}$. /* Update History Window. */
8 **if** $|W| \geq \varpi$ **then**
9 $W \leftarrow W \setminus \{(x_{t-\varpi}, y_{t-\varpi})\}$.
10 Compute CDT metric s_t. /* Active Step. */
11 Apply CDT $g_t \leftarrow \vartheta(s_1, \ldots, s_t)$.
12 **if** $g_t \geq h$ **then** /* Change Detection Check. */
13 Estimate Real Change Time t_r.
14 $\mathcal{T} \leftarrow \{(x_{\bar{t}}, y_{\bar{t}}) \in W : \bar{t} \geq t_r\}$. /* Novel Samples. */
15 *[Optional] Condense $\mathcal{T}$.*
16 Update $\mathcal{D}$ with $\mathcal{T}$.

6.3.2 Active Update: Active Tiny kNN

The Active Tiny kNN, whose pseudocode is shown in Algorithm 6.4, relies on a Change Detection Test (CDT) ϑ to detect changes in the data generation process $\mathcal{P}$. The core of this algorithm is the ability to adapt the classifier $\mathcal{K}$'s training set $\mathcal{T}$ only after the detection of a concept drift. In addition, the Active Tiny kNN allocates space for a history window W of size ϖ, being ϖ a parameter of the algorithm (described in the sequel).

In more detail, for each sample (x_t, y_t) provided by $\mathcal{P}$ at time instant t, the Active Tiny kNN predicts the label $\hat{y}_t = \mathcal{D}(x_t)$ and, when the supervised information y_t is available, the active update is activated (Algorithm 6.4, Lines 7–16).

At first, it adds the pair (x_t, y_t) to the history window W and discards the oldest pair if the window already contains ϖ pairs (Algorithm 6.4, Lines 7–9). After that, the Active Tiny kNN computes the figure of merit s_t (at time t) and applies the CDT decision function ϑ to inspect for changes in $\mathcal{P}$ (Algorithm 6.4, Lines 10–11), i.e., $g_t = \vartheta(s_1, \ldots, s_t)$. In the most general situation, the computation of g_t at time t takes into account all the figures of merits computed from $t = 1$. A change is detected in the data-generation process $\mathcal{P}$ when g_t overcomes the detection threshold h, being h a parameter of the algorithm (Algorithm 6.4, Line 12).

Once a change is detected, the adaptation stage starts (Algorithm 6.4, Lines 13–16). In the first place, it estimates the time t_r the change occurred at (e.g., with a Change Point Method). After that, it discards from the history window W all the samples older than the estimated change time t_r. The updated history window W (optionally condensed through Algorithm 6.2) becomes the new $\mathcal{K}$'s training set $\mathcal{T}$.

It is noteworthy to point out that the memory footprint of the Active Tiny kNN is bounded over time since it requires to store the training set $\mathcal{T}$ and history window W of

at most ϖ samples (the CDT memory footprint can be neglected). Moreover, since the adaptation stage modifies the knowledge set $\mathcal{T}$ only through copies of the (at most whole) history window W, the total memory footprint cannot overcome twice the memory of the history window W, i.e., that of 2ϖ samples.

Although the solution accepts as input any CDT ϑ, in the context of this work, ϑ is the well-known and theoretically grounded CUSUM algorithm (Page, 1954) in its generalized version (Lorden et al., 1971), monitoring the accuracy of the Active Tiny kNN over time. As a consequence, any change in the data-generation process $\mathcal{P}$ is assumed to reflect on the $\mathcal{K}$ classification accuracy.

The generalized CUSUM CDT is designed as follows. Let υ_0 be the stationary classification accuracy (estimated on the first ξ supervised samples in the testing stage, being ξ parameter of the Active Tiny kNN algorithm). A Bernoulli distribution with parameter υ_0 and, in turn, a Binomial distribution with parameters υ_0 and n (with n size of the batches on which the accuracy is computed in the following) model our scenario in stationary conditions. The figure of merit of the CUSUM CDT is the likelihood ratio of the probability distributions modeling the scenario after and before the change, i.e.:

$$s_t = \ln \frac{p_{\upsilon_1}(\zeta_t)}{p_{\upsilon_0}(\zeta_t)}, \tag{6.2}$$

where υ_1 represents the classification accuracy after the change and ζ_t the realization of the Binomial distribution at time t, i.e., the accuracy on the n supervised samples arrived before time t.[11]

Since the value of υ_1 is a priori unknown, the generalized version of the CUSUM algorithm employs a set Υ_1 containing a grid of possible accuracies υ_1 after change equally spaced from 0 to 1, except for the neighborhood of υ_0.[12] The resulting decision function ϑ is:

$$g_t = \vartheta(s_1, \ldots, s_t) = \max_{1 \le j \le t} \sup_{\upsilon_1 \in \Upsilon_1} S_j^t(\upsilon_1), \tag{6.3}$$

where

$$S_j^t(\upsilon_1) = \sum_{i=j}^{t} s_i, \tag{6.4}$$

represents the sum of the log-likelihood ratio s_ts from time j to time t.

Assuming the parameter n large enough, the considered Binomial distribution can be approximated as a Normal one with mean $n\upsilon_0$ and variance $n\upsilon_0(1-\upsilon_0)$. Hence, the log-likelihood ratio s_t in Equation 6.2 then becomes:

$$s_t = \frac{\upsilon_1\bar{\upsilon_1} - \upsilon_0\bar{\upsilon_0}}{2n\upsilon_0\bar{\upsilon_0}\upsilon_1\bar{\upsilon_1}}\zeta_t^2 + \frac{\bar{\upsilon_0}\bar{\upsilon_1}}{\bar{\upsilon_0}\bar{\upsilon_1}}\zeta_t + \frac{n(\upsilon_0\bar{\upsilon_1} - \upsilon_1\bar{\upsilon_0})}{2\bar{\upsilon_0}\bar{\upsilon_1}} + \ln\sqrt{\frac{\upsilon_0\bar{\upsilon_0}}{\upsilon_1\bar{\upsilon_1}}}, \tag{6.5}$$

where $\bar{\upsilon_0} = 1 - \upsilon_0$ and $\bar{\upsilon_1} = 1 - \upsilon_1$.

[11] Although the Active Tiny kNN algorithm is general enough to deal with any CDT, in the described CUSUM case with Binomial distribution of size n, the CDT figure of merit is not computed for every supervised samples, but every n. As a consequence, all the $n-1$ s_t values before a window of size n is full are set to zero.

[12] The cardinality of Υ_1, i.e., the number of tested values υ_1, is a parameter of the Active Tiny kNN.

Algorithm 6.5: The Hybrid Tiny kNN.

Input: Feature Extractor $\varsigma \circ \varrho$, CDT ϑ, Training Set $\mathcal{T}$.
Parameters : Maximum $\mathcal{T}$ Size ϖ, CDT threshold h.

Compute $\mathcal{T} \leftarrow \bar{\mathcal{T}}$ with Algorithm 6.2. /* Condense $\mathcal{T}$. */
Initialize the k-NN classifier $\mathcal{K}$ with $\varsigma \circ \varrho(\mathcal{T})$.
Define $\mathcal{D} = \mathcal{K} \circ \varsigma \circ \varrho$.
/* Loop over samples arriving at time t. */
foreach $(x_t, y_t) \sim \mathcal{P}, t = 1, 2, \ldots$ **do**
    Predict $\hat{y}_t \leftarrow \mathcal{D}(x_t)$.
    if $\hat{y}_t \neq y_t$ **then** /* Passive Update. */
        $\mathcal{T} \leftarrow \mathcal{T} \cup (x_t, y_t)$
        if $|\mathcal{T}| \geq \varpi$ **then**
            $t_{min} \leftarrow \min_{\bar{t}} \{(x_{\bar{t}}, y_{\bar{t}}) \in \mathcal{T}\}$
            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(x_{t_{min}}, y_{t_{min}})\}$.
        Update $\mathcal{D}$ with $\mathcal{T}$.
    Compute CDT metric s_t. /* Active Step. */
    Apply CDT $g_t \leftarrow \vartheta(s_1, \ldots, s_t)$.
    if $g_t \geq h$ **then** /* Change Detection Check. */
        Estimate Real Change Time t_r.
        $\mathcal{T} \leftarrow \{(x_{\bar{t}}, y_{\bar{t}}) \in \mathcal{T} : \bar{t} \geq t_r\}$. /* Novel Samples. */
        [Optional] Condense $\mathcal{T}$.
        Update $\mathcal{D}$ with $\mathcal{T}$.

As a final remark, the CUSUM CDT is also endowed with the ability to estimate the change time t_r (and, if desired, of the parameter υ_1 after the change). The estimated change time t_r is indeed the index $\tilde{j}$ maximizing the decision function ϑ in Eq. (6.3), i.e.:

$$\left(\tilde{j}, \tilde{\upsilon_1}\right) = \arg\max_{1 \leq j \leq t} \sup_{\upsilon_1 \in \Upsilon_1} S_j^t(\upsilon_1). \tag{6.6}$$

6.3.3 Hybrid Tiny kNN: Integrating Condensing-in-Time and Active Tiny kNN

The core of the proposed hybrid update is to integrate the "condensing-in-time" ability of the passive update with the capability to quickly adapt to changes by discarding obsolete knowledge of the active one.

In more detail, the (CIT) passive update continuously adapts $\mathcal{T}$ when supervised information is available, regardless a concept drift occurred (or not). This ability comes at the expense of two weak points. First, there is (in principle) no bound on the memory occupation, although two solutions have been suggested to mitigate the problem. Second, when a change occurs, the passive update does not discard the obsolete knowledge present in $\mathcal{T}$, i.e., samples generated by $\mathcal{P}$ before the concept drift occurred.

On the contrary, the active adaptation provides a bound on the memory occupation (i.e., twice the history window size W) and, in turn, the required computation. However, similarly to the other active approaches present in the literature (Baena-Garcıa et al., 2006; Disabato and Roveri, 2019; Wang et al., 2020b), the effectiveness of the active adaptation

phase is strictly related to the ability to promptly detect the concept drift in $\mathcal{P}$.

The proposed hybrid update aspires at compensating the weak points of passive and active updates by integrating the "condensing-in-time" solution described in Algorithm 6.2 with the CUSUM-based CDT detailed in Section 6.3.2. The resulting algorithm, namely the Hybrid Tiny kNN, is shown in Algorithm 6.5. Here, the inputs and the initialization are the same as Active Tiny kNN. The only difference resides in the fact that the Hybrid Tiny kNN does not allocate a history window, but it relies on the training set $\mathcal{T}$ (whose size is bounded by ϖ) as history window.

Similarly to the algorithms it derives from, the Hybrid Tiny kNN predicts the label $\hat{y}_t = \mathcal{D}(x_t)$ and, when the supervised information y_t is made available, it carries out both a passive (Algorithm 6.5, Lines 6–11) and an active update (Algorithm 6.5, Lines 12–18), for each sample (x_t, y_t) generated by $\mathcal{P}$ at time instant t. Although the passive update is equal to that of the "condensing-in-time" algorithm, the active one requires to take into account the effects of the passive updates, which are supposed to increase the classification capability of the algorithm over time (until the accuracy of Hybrid Tiny kNN reaches its maximum value). Consequently, the CUSUM CDT is slightly modified in its set Υ_1, which contains only values that are smaller than υ_0, i.e., the accuracy estimated on an initial window of data. In this way, the hybrid update does not detect as concept drift the increases in the accuracy brought by the passive update (hence focusing on changes inducing a drop in the accuracy). Moreover, the adaptation phase triggered by the CDT involves directly the knowledge set $\mathcal{T}$ of the classifier $\mathcal{K}$, where samples older than the estimated time of change t_r are discarded (Algorithm 6.5, Lines 15–18).

Summing up, the hybrid update continuously adapts $\mathcal{T}$ over time thanks to the passive adaptation, hence avoiding the risk of non-detecting changes due to false-negative detections of the CDT. At the same time, the active adaptation present in the hybrid update can quickly discard obsolete knowledge when a change is detected and set a bound on the memory footprint of $\mathcal{T}$. All these aspects will be evaluated in Section 9.4.5.

6.4 Discussion: Parameters Choice and Limitations

This section firstly proposes a few suggestions about the choice of the proposed algorithms' hyper-parameters, then there is a discussion about the major limitation of those algorithms, in addition to the discussion about changes that have a small impact on DLM accuracy (see Section 5.3.3).

In both Active Tiny kNN and Hybrid Tiny kNN algorithms, the choice of the threshold h is application-dependent. More in detail, such value should be a trade-off between the number of expected false positives and negatives. On the one hand, a small value for h can result in a high number of false positives, i.e., detecting changes that have never occurred and activating an unnecessary adaptation step. Such adaptation might remove non-obsolete knowledge from the training set $\mathcal{T}$, hence potentially impacting the algorithm in the subsequent steps. Moreover, a high rate of false positives is undesirable in critical scenarios where each detection has an impact (e.g., airport surveillance where a detection causes a block of the security checks until the *issue* has been solved). On the other hand, a too high value of h will result in an increased number of false negatives, i.e., not detecting a change that has occurred. Consequently, the obsoleted knowledge is kept, whereas the fresh and novel one might not, with impacts on the algorithms.

The choice of the history window size in Active Tiny kNN or the knowledge set $\mathcal{T}$ size in Hybrid Tiny kNN ϖ, as well as the maximum number of training samples p in CIT algorithm, can be tuned according to the application. However, the choice is mainly based on the technological constraints of the IoT unit the algorithm is deployed on. As a rule of thumb, ϖ or p size can be the maximum available according to the memory capacity $\bar{m}$ of the considered IoT unit, making sure that there is enough room for all the algorithms running on the device.

The algorithms presented in this chapter (but also that in Section 5.3) are all supervised. This assumption can be a major limitation in all those scenarios where the supervision is rarely provided or it is not available at all. In both cases, defining a CDT ϑ that relies on an unsupervised metric can alleviate this issue. As an example, instead of defining the CDT ϑ on the classification accuracy, one can model the spatial representation of the classes in stationary conditions (e.g., through clustering algorithms to obtain a compressed representation through the centroids and properly defined volumes enclosing each class data) and then measure when such representation starts to drift. However, supervision is highly suggested to correctly handle the new representation once the CDT detects a change. These aspects are left for future work (see Section 12.1).

Part III

Deep Wide Tiny Machine Learning

CHAPTER 7

Distribute Computation over Heterogeneous Internet of Things Units

Chapters 5 and 6 addressed the problem of defining a Tiny Machine Learning algorithm able to be executed on a single IoT unit (as well as introducing solutions for the on-device learning). A totally different approach to the problem aims at distributing the computation of a Deep Learning Model (DLM) across several IoT units (that are thus organized into an IoT system) (Disabato et al., 2021b). The advantage of this approach is that a DLM that cannot be executed on a single IoT unit due to its memory, computation, or energy requirements, can instead be executed on a group of IoT units. Each IoT unit supports the execution of part of the DLM (e.g., one or more of its layers) according to its memory, computation, and energy capacity, and then forward the computed results to the other IoT unit(s) executing the subsequent part of the DLM.

A methodology aiming at addressing this problem usually models the *distributed* DLM as a directed graph spanning over the network of IoT units (where each node corresponds to a subtask of the DLM on the IoT unit executing it and each arc the sequence of those subtasks). Such a methodology takes into accounts the following aspects. First of all, the original DLM (or more generally, a group of DLMs to be executed at the same time), its requirements, and all the possible subtasks that can be defined within it (typically, the division is made at layer-level, but further splits can be designed as well). Second, the memory capacity, the available computation, and the energy budget of all the available IoT units within the IoT system. Third, the methodology has to take into account the network architecture: the position and distance of the IoT units along with the connectivity

technology employed to transmit the information between IoT units (not only in terms of transmission speed and range but also possible failures and retransmissions to be more robust). Finally, the last aspect is the deployment of the computed directed graph and its possibility to adapt to new operating conditions (e.g., due to concept drift in the data generation process, the introduction of new DLMs to be executed, changes in the network topology because of IoT units addition, IoT units running out of energy).

The problem is formalized in Section 4.3, with a complement in Section 7.1. The methodology (Disabato et al., 2021b) is then described in the remaining Sections (tailored to specific cases). It addresses the problem of finding the optimal placement of the C DLMs layers on the N IoT units in order to minimize the latency in transmitting decisions about the inputs gathered by the C sources to the target unit f. Finally, Section 10.1 experimental evaluates the methodology.

7.1 Dealing With Early-Exit Deep Learning Models

Section 4.3 formalizes the most general case of a system of (possibly heterogeneous) IoT units, with the most general definition of the DLMs to be mapped on it. In this Section, such formalization is extended to consider all those DLMs whose processing path depend on the information content, e.g., the Adaptive (Bolukbasi et al., 2017) or Gate-Classification CNNs (Disabato and Roveri, 2018). In these DLMs, the classification completes as soon as enough confidence is achieved within the processing path, thus the execution of the remaining layers is aborted.

To achieve this goal, Early-Exit DLMs (EX-DLMs) are endowed with intermediate exit points, creating multiple paths within the DLM, each of which is characterized by a probability of being traversed. More formally, given a M_u-layer EX-DLM, let $p_{u,j} \in [0,1]$ be the probability that the j-th layer of the u-th DLM processes the input image[13] and let $g_{u,j} \in [0,1]$, for each $u \in \mathbb{N}_C$ and $j \in \{1,\dots,M_u\}$, be the probability that the computation ends at layer j of the u-th EX-DLM is

$$g_{u,j} = \begin{cases} p_{u,j} - p_{u,j+1} & \text{if } j < M_u \\ 1 - \sum_{v=1}^{M_u-1} g_{u,v} & \text{if } j = M_u \end{cases}. \tag{7.1}$$

7.2 Distributing Deep Learning Models over an IoT System as a Quadratic Optimization Problem

The methodology aiming at distributing the DLMs over an IoT system is formulated as an optimization problem that relies on the $C \cdot N \cdot M$ variables $\alpha_{u,i,j}$ defined as:

$$\alpha_{u,i,j} = \begin{cases} 1 & \text{if IoT unit } i \text{ executes layer } j \text{ of DLM } u \\ 0 & \text{otherwise} \end{cases}, \tag{7.2}$$

for each $u \in \mathbb{N}_C$, for each $i \in \mathbb{N}_N$ and for each $j \in \mathbb{N}_M = \{1,\dots,M\}$, being $M = max\{M_1,\dots,M_u\}$ the maximum number of layers among the considered C DLMs (i.e., the maximum depth of all DLMs).

[13] In the case of the mentioned Early-Exit CNNs, the probabilities $p_{u,j}$s can be estimated during their learning (Disabato and Roveri, 2018; Teerapittayanon et al., 2016) or on a validation set immediately after.

Table 7.1: *Details about the IoT units and Deep Learning Models used in Chapter 7 and its corresponding experimental evaluation (Section 10.1).*

(a) *The memory demand m_j, the computational load c_j, and the memory K_j required to store the intermediate representations of the 5-layer DLM and the 6-layer EX-DLM, with a 32-bit data type and the Early-Exit layer marked with an asterisk. In that layer, K_j represents the dimensions of the representation sent to the layer $j+1$ when the classification is not taken at layer j.*

	Layer (j)	m_j (KB)	c_j (10^6 mult.)	K_j (KB)
s	Source (Image $28 \times 28 \times 3$)	-	-	9.41
$L1_1$	5x5 convolution, 64 filters	19.20	3.76	200.70
$L1_2$	2x2 max pooling, stride 2	-	0.05	50.18
L2*	gc1 (3-layer dense 384,192,10)	19 570.18	4.89	50.18
$L3_1$	5x5 convolution, 64 filters	409.60	20.07	50.18
$L3_2$	2x2 max pooling, stride 2	-	0.01	12.54
L4	fully-connected 384	4 816.90	1.20	1.54
L5	fully-connected 192	294.91	0.07	0.77
L6	fully-connected 10	7.68	0.002	0.04

(b) *The memory demand m_j, the computational load c_j, and the memory K_j required to store the intermediate representations of the AlexNet and its EX-DLM variant, with a 32-bit data type and the Early-Exit layer marked with an asterisk. In that layer, K_j represents the dimensions of the representation sent to the layer $j+1$ when the classification is not taken at layer j.*

	Layer (j)	m_j (KB)	c_j (10^6 mult.)	K_j (KB)
s	Source (Image $227 \times 227 \times 3$)	-	-	618.35
1_1	11x11 convolution, 96 filters, stride 4	139.78	105.42	1161.60
1_2	3x3 max pooling, stride 2	-	0.31	279.94
2_1	5x5 convolution, 256 filters	1 229.82	223.95	746.50
2_2	3x3 max pooling, stride 2	-	0.39	173.06
3^*	gc1(3-layer dense 128,64,2)	22 185.22	5.55	173.06
4	3x3 convolution, 384 filters	3 540.48	149.52	259.58
5	3x3 convolution, 384 filters	2 655.74	112.14	259.58
6_1	3x3 convolution, 256 filters	1 770.50	74.76	173.06
6_2	3x3 max pooling, stride 2	-	0.08	36.86
7	fully-connected 4096	151 011.39	37.75	16.38
8	fully-connected 4096, 2	67 158.02	16.78	0.01

(c) *The maximum memory usage $\bar{m}_i$ (defined as half the available RAM), and the million (10^6) multiplications per second e_is (defined as a tenth of the clock cycles performed in one second) of the considered IoT units.*

	Node (i)	$\bar{m}_i$ (KB)	e_i (10^6 mult.)
S1	STM32H7 (480MHz ARM Cortex M7, no OS)	512	48
B1	BeagleBone AI (1.5GHz Dual-core ARM Cortex A15)	524 288	300
O1	OrangePi Zero (1.2GHz H2 Quad-core ARM Cortex-A7)	131 072	480
R1	Raspberry Pi 3B+ (1.4GHz Quad-core ARM Cortex-A53)	524 288	560

Without any loss of generality, the distances d_{i_1,i_2}, for each $i_1, i_2 \in V$, can be precomputed, allowing us to define an integer quadratic optimization problem on variables $\alpha_{u,i,j}$s. As detailed in Section 4.3, $\{s_1, \ldots, s_C\}$ and f do not participate in the optimization prob-

lem since their task is to acquire inputs and receive the final classification, respectively. This assumption can be easily removed by considering additional IoT computing units in the same positions of s_is and f.

The objective function to be minimized models the latency in making a decision by the C DLMs placed on the IoT units, defined as the time occurring between inputs acquisition by sensor unit s_us (size $K_{u,s}$) and final classifications K_{u,M_u}s are transmitted to unit f:

$$\sum_{u=1}^{C}\sum_{i=1}^{N}\sum_{k=1}^{N}\sum_{j=1}^{M-1}\alpha_{u,i,j}\cdot\alpha_{u,k,j+1}\cdot p_{u,j+1}\cdot\frac{K_{u,j}}{\rho}\cdot d_{i,k}+\sum_{i=1}^{N}t_i^{(p)}+t_s+t_f, \tag{7.3}$$

such that

$$\forall i\in\mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\leq L \tag{7.4}$$

$$\forall i\in\mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot m_{u,j}\leq \bar{m}_i \tag{7.5}$$

$$\forall i\in\mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot c_{u,j}\leq \bar{c}_i \tag{7.6}$$

$$\forall u\in\mathbb{N}_C,\forall j\in\mathbb{N}_M \qquad \sum_{i=1}^{N}\alpha_{u,i,j}=\begin{cases}1 & \text{if } j\leq M_u\\ 0 & \text{otherwise}\end{cases} \tag{7.7}$$

and where

$$t_s = \sum_{u=1}^{C}\sum_{i=1}^{N}\alpha_{u,i,1}\cdot p_{u,1}\cdot\frac{K_{u,s}}{\rho}\cdot d_{s,i} \tag{7.8}$$

$$t_i^{(p)} = \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot p_{u,j}\cdot\frac{c_{u,j}}{e_i} \tag{7.9}$$

$$t_f = \sum_{u=1}^{C}\sum_{i=1}^{N}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot g_{u,j}\cdot\frac{K_{u,M_u}}{\rho}\cdot d_{i,f} \tag{7.10}$$

The objective function in Eq. (7.3) comprises four different components of the latency:

(i) The source time t_s, defined in Eq. (7.8), required to transmit the inputs from the sources s_us to the IoT units executing the first layer of the DLMs. Although the first layer is always reached, i.e., $p_{u,1}=1$ for each $u\in\mathbb{N}_C$, the term $p_{u,1}$ has been added to Eq. (7.8) to provide homogeneity in the formalization.

(ii) The transmission time of intermediate representations among the IoT units processing the DLM layers. More precisely, the transmission time of the intermediate representation of the j-th layer of the u-th DLM from unit i to k is

$$t_t=\frac{K_{u,j}}{\rho}\cdot d_{i,k}, \tag{7.11}$$

where ρ is the data-rate of the considered transmission technology and $d_{i,k}$ is the hop-distance between IoT units i and k as defined in Section 4.3. In Eq. (7.3) the transmission time is weighted by the probability $p_{u,j+1}$ that layer $j+1$ is executed right after layer j.

(iii) The processing time $t_i^{(p)}$ of the DLM layers on the IoT units. Specifically, the processing time of layer j of DLM u on the i-th IoT unit is approximated as the ratio between the computational demand $c_{u,j}$ that layer requires and the number of multiplications e_i the IoT unit i carries out in one second[14]. In Eq. (7.9), the processing time is weighted by the probability $p_{u,j}$ that the layer j of DLM u is executed.

(iv) The sink time t_f required to transmit the final classification K_{u,M_u}, for each $u \in \mathbb{N}_C$, from the IoT units taking these decisions to the target unit f. It is noteworthy to point out that Eq. (7.10) takes into account all feasible output paths from node i to the target unit f, suitably weighted by the probability $g_{u,j}$ that the classification is made at layer j of DLM u in execution on IoT unit i.

The constraint in Eq. (7.4) ensures that each IoT unit contains at most L layers, being L an additional user-defined model hyper-parameter. In particular, when $L = 1$, at most one layer can be assigned to an IoT unit, while $L > 1$ implies that up to L layers (also belonging to different DLMs) can be assigned to a particular IoT unit. The constraints in Eq. (7.5) and (7.6) are meant to take into account the technological limits about memory usage and computational load characterizing each IoT unit. Finally, the constraint in Eq. (7.7) ensures that each layer j, for each $j \in \mathbb{N}_M$, is assigned to exactly one node and, at the same time, manages the possibility that the C DLMs might be characterized by a different number $M_u \leq M$ of layers, for each $u \in \mathbb{N}_C$. In fact, in those cases (i.e., when $M_u < M$), the unneeded $\alpha_{u,i,j}$s are set to 0.

When the j_1-th layer of DLM u_1 and the j_2-th layer of DLM u_2 are shared between the two DLMs, the following constraint is added to the optimization problem

$$\forall i \in \mathbb{N}_N \qquad \alpha_{u_1,i,j_1} = \alpha_{u_2,i,j_2}, \tag{7.12}$$

to ensure that the shared layer is placed on the same IoT unit. In addition, the constraints on the maximum number of layers placed on a IoT unit - Eq. (7.4) - and the memory usage and computational load constraints - Eqs (7.5) and (7.6) - are modified as follows to count the shared layer only once:

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M} \alpha_{u,i,j} \leq L + \alpha_{u_2,i,j_2}, \tag{7.13}$$

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M} \alpha_{u,i,j} \cdot m_{u,j} \leq \bar{m}_i + \alpha_{u_2,i,j_2} \cdot m_{u_2,j_2}, \tag{7.14}$$

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M} \alpha_{u,i,j} \cdot c_{u,j} \leq \bar{c}_i + \alpha_{u_2,i,j_2} \cdot c_{u_2,j_2}. \tag{7.15}$$

[14]The e_is encompass the number of available cores, the type of pipeline such cores implement to approach one operation per clock cycle, the presence or not of a GPU allowing to parallelize DLM operations (e.g., the convolutions) and all the delays resulting from the processing system and memory management.

If a layer is shared among k DLMs, the Eqs. (7.13), (7.14), and (7.15) need to take into account $k-1$ out of the k variables corresponding to the shared layer.

The considered class of optimization problems, i.e., the integer quadratic programs, is NP-complete. More specifically, since the $\alpha_{u,i,j}$s are binary variables, it is possible to convert it to a binary linear program, which is one of Karp's 21 NP-complete problems (Karp, 1972).

The optimization problem outcome is the optimal placement $\alpha_{u,i,j}$s of the C DLMs' layers to the N IoT units minimizing the delay in making a classification. In the event that the optimization provides more than one solution, the optimal placement is any feasible solution with minimal latency. More advanced mechanisms could be considered, e.g., selecting the configuration characterized by the lowest energy consumption in transmission or computation.

7.3 Distributing a Single DLM over an IoT System

When the number of DLMs is equal to one, i.e., $C = 1$, the optimization problem variables formalization in Eq. (7.2) simplifies in $N \cdot M$ binary variables $\alpha_{i,j}$ to determine whether layer j of the DLM is assigned to unit i of the IoT system, i.e.,

$$\alpha_{i,j} = \begin{cases} 1 & \text{if IoT unit } i \text{ executes DLM layer } j \\ 0 & \text{otherwise} \end{cases}, \tag{7.16}$$

for each $i \in \mathbb{N}_N$ and $j \in \mathbb{N}_M$. Moreover, the objective function in Eq. (7.3) modelling the latency in making a decision to be minimized is reformulated as:

$$\sum_{i=1}^{N} \sum_{k=i}^{N} \sum_{j=1}^{M-1} \alpha_{i,j} \cdot \alpha_{k,j+1} \cdot \frac{K_j}{\rho} \cdot d_{i,k} + \sum_{i=1}^{N} t_i^{(p)} + t_s + t_f, \tag{7.17}$$

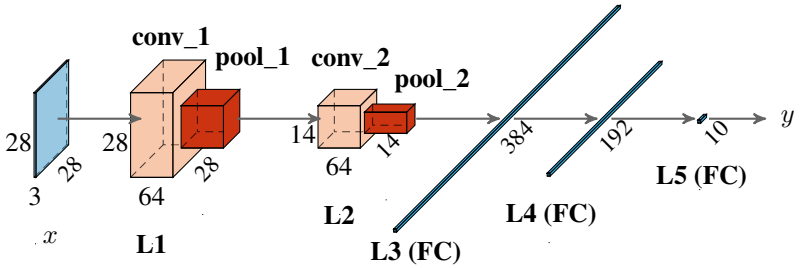
where the probability of executing each layer is omitted being equal to 1 for all of them. Then, the constraints in Eqs. (7.4), (7.5), (7.6) and (7.7) become:

$$\forall i \in \mathbb{N}_N \qquad \sum_{j=1}^{M} \alpha_{i,j} \leq L, \tag{7.18}$$

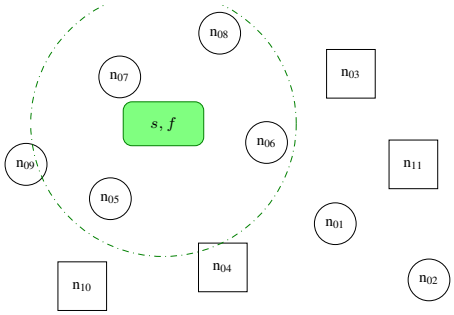
$$\forall i \in \mathbb{N}_N \qquad \sum_{j=1}^{M} \alpha_{i,j} \cdot m_j \leq \bar{m}_i, \tag{7.19}$$

$$\forall i \in \mathbb{N}_N \qquad \sum_{j=1}^{M} \alpha_{i,j} \cdot c_j \leq \bar{c}_i, \tag{7.20}$$

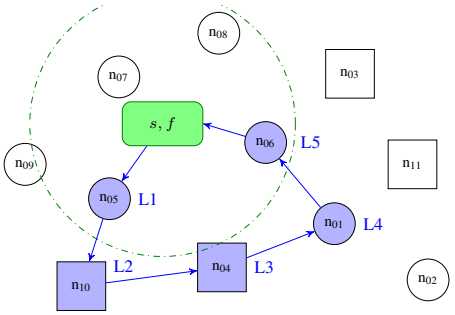
$$\forall j \in \mathbb{N}_M \qquad \sum_{i=1}^{N} \alpha_{i,j} = 1, \tag{7.21}$$



(a) *The architecture of the 5-layer DLM detailed in Table 7.1a, where x is the input image.*



(b) *An example of IoT system with STM32H7 (circles) and Raspberry 3B+ (squares). The source s and the sink f share the same IoT unit. The dotted circle refers to the transmission range, equal for all the IoT units.*



(c) *An example of the methodology outcome where the layers L1,..., L5 of the 5-layer DLM in Figure 7.1a are placed on the IoT units of the system shown in Figure 7.1b, when setting $L = 1$.*

	n_{01}	n_{02}	n_{03}	n_{04}	n_{05}	n_{06}	n_{07}	n_{08}	n_{09}	n_{10}	n_{11}
L1	0	0	0	0	**1**	0	0	0	0	0	0
L2	0	0	0	0	0	0	0	0	0	**1**	0
L3	0	0	0	**1**	0	0	0	0	0	0	0
L4	**1**	0	0	0	0	0	0	0	0	0	0
L5	0	0	0	0	0	**1**	0	0	0	0	0

(d) *The variables $\alpha_{u,i,j}$s representing the methodology outcome for the solution shown in Figure 7.1c, with $u = 1$.*

Figure 7.1: *Distributing a single 5-layer DLM (i.e., $C = 1$) over an IoT system.*

whereas

$$t_s = \sum_{i=1}^{N} \alpha_{i,1} \cdot \frac{K_s}{\rho} \cdot d_{s,i}, \tag{7.22}$$

$$t_i^{(p)} = \sum_{j=1}^{M} \alpha_{i,j} \cdot \frac{c_j}{e_i}, \tag{7.23}$$

$$t_f = \sum_{i=1}^{N} \alpha_{i,M} \cdot \frac{K_M}{\rho} \cdot d_{i,f}, \tag{7.24}$$

account for the transmission time between the source s and the IoT unit running the first layer of the DLM, the processing time on all unit is, and the transmission time between the unit running the M-th layer of the DLM and the sink f, respectively.

In this configuration, the methodology has been applied as an example to the 5-layer DLM described in Figure 7.1a, characterized by $M = 5$ layers and whose details are in Table 7.1a. The considered IoT system is the one shown in Figure 7.1b comprising $N = 11$ IoT units and being s and f the same unit. The IoT units belong to two different technological families, i.e., STM32H7 (round nodes) and Raspberry Pi 3B+ (squared nodes), whose memory $\bar{m}_i$ and computational $\bar{c}_i$ constraints are detailed in Table 7.1c. An example of the optimization problem outcome in this technological scenario with $L = 1$ is given in Figure 7.1c, whose corresponding $\alpha_{i,j}$s are detailed in Figure 7.1d. In the optimal placement, involving three STM32H7 units (nodes n_{05}, n_{01}, and n_{06}) and two Raspberry Pi 3B+ (nodes n_{10}, and n_{04}), the layer $L3$ of the DLM has been assigned to a Raspberry Pi 3B+ IoT unit (i.e., n_{04}) since its execution on STM32H7 would violate the memory constraint.

7.4 Distributing a Single EX-DLM over an IoT System

This configuration refers to the case where a single Early-Exit Deep Learning Model (EX-DLM) has to be placed on the IoT system, i.e., $C = 1$.

More specifically, the problem variables are simplified into $N \cdot M$ binary variables $\alpha_{i,j}$, as defined in Eq. (7.16). The $p_{u,j}$s and $g_{u,j}$s are simplified as p_j and g_j, for each $j \in \mathbb{N}_M$, defining the probabilities that layer j is executed and that the final classification is made at layer j (i.e., the direct path from j to the sink is traversed), respectively.

The objective function modeling the latency in decision making defined in Eq (7.3) is tailored as follows:

$$\sum_{i=1}^{N}\sum_{k=i}^{N}\sum_{j=1}^{M-1}\alpha_{i,j}\cdot\alpha_{k,j+1}\cdot p_{j+1}\cdot\frac{K_j}{\rho}\cdot d_{i,k}+\sum_{i=1}^{N}t_i^{(p)}+t_s+t_f, \tag{7.25}$$

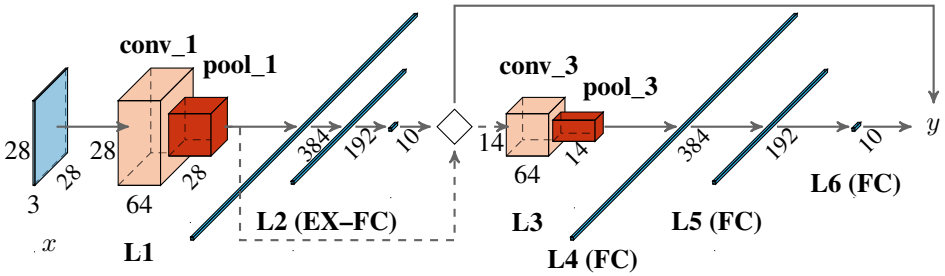
with constraints as in Eqs. (7.18), (7.19), (7.20), and (7.21), and where the source time t_s, the processing time $t_i^{(p)}$, and the sink time t_f have been modified as follows:

$$t_s = \sum_{i=1}^{N}\alpha_{i,1}\cdot p_1\cdot\frac{K_s}{\rho}\cdot d_{s,i}, \tag{7.26}$$

$$t_i^{(p)} = \sum_{j=1}^{M}\alpha_{i,j}\cdot p_j\cdot\frac{c_j}{e_i}, \tag{7.27}$$

$$t_f = \sum_{i=1}^{N}\sum_{j=1}^{M}\alpha_{i,j}\cdot g_j\cdot\frac{K_M}{\rho}\cdot d_{i,f}. \tag{7.28}$$

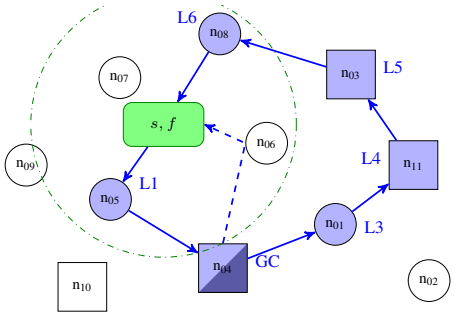
A 6-layer EX-DLM is shown, as an example, in Figure 7.2a and detailed in Table 7.1a. This EX-DLM has $M = 6$ and the Early Exit at layer $j = 2$, with a probability $\nu = 0.99$ of taking the final classification. Hence, $p_1 = p_2 = 1$ and $p_3 = p_4 = p_5 = 0.01$, with the g_js different from zero only at the Early-Exit ($g_2 = 0.99$) and the last layer ($g_6 = 0.01$).



(a) *The architecture of the 6-layer EX-DLM detailed in Table 7.1a, where x is the input image. The Early-Exit (composed of three fully-connected layers) can either provide the final output (solid line from the "decision" diamond) or further exploring the pipeline (dashed path, highlighting that the forwarded features are those in input to the Early-Exit layer).*

	p_j	g_j
L1	1.00	0.00
L2	1.00	0.99
L3	0.01	0.00
L4	0.01	0.00
L5	0.01	0.00
L6	0.01	0.01

(b) *The values of p_j and g_j, for each $j \in \mathbb{N}_M$, of the 6-layer EX-DLM architecture shown in Figure 7.2a.*



(c) *The methodology outcome with a 6-layer EX-DLM in Figure 7.2a on the IoT system shown in Figure 7.1b, with $L = 1$.*

Figure 7.2: *Distributing a single 6-layer EX-DLM over an IoT System. Since $d_{n_{04},f} = 2$, n_{04} requires an intermediate hop, i.e., the node n_{06}, to send the final classification.*

Figure 7.2 shows the methodology outcome in this configuration on the IoT system already described in Figure 7.1b. The methodology outcome is particularly interesting, showing that the Early-Exit layer ($j = 2$), being particularly demanding in terms of memory, is assigned to the Raspberry Pi 3B+ n_{04} unit. When enough confidence is achieved at Early-Exit ($j = 2$), the decision is directly sent from n_{04} to the sink f, otherwise, the processing is forwarded from n_{04} to n_{01} to complete the processing up to n_{08}. As a final observation, it is worth noting that the distance between the Early-Exit ($j = 2$) and the sink f is equal to 2. Thus this intermediate classification is delivered through the node n_{06}.

7.5 Distributing DLMs with Shared Layers over an IoT System

An interesting application scenario is the one with multiple DLMs, either without or with shared processing layers. In particular, the DLMs do not employ early exits, hence $p_{u,j} = 1$, for each $u \in \mathbb{N}_C$ and $j \in \{1, \ldots, M_u\}$.

In this configuration, the objective function modeling the latency in decision making

becomes:

$$\sum_{u=1}^{C}\sum_{i=1}^{N}\sum_{k=i}^{N}\sum_{j=1}^{M-1}\alpha_{u,i,j}\cdot\alpha_{u,k,j+1}\cdot\frac{K_{u,j}}{\rho}\cdot d_{i,k}+\sum_{i=1}^{N}t_i^{(p)}+t_s+t_f, \tag{7.29}$$

with constraints defined in Eqs. (7.4), (7.5), (7.6) and (7.7), and where the source time t_s, the processing time $t_i^{(p)}$, and the sink time t_f are modified as follows:

$$t_s = \sum_{u=1}^{C}\sum_{i=1}^{N}\alpha_{u,i,1}\cdot\frac{K_{u,s}}{\rho}\cdot d_{s,i}, \tag{7.30}$$

$$t_i^{(p)} = \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot\frac{c_{u,j}}{e_i}, \tag{7.31}$$

$$t_f = \sum_{u=1}^{C}\sum_{i=1}^{N}\alpha_{u,i,M_u}\cdot\frac{K_{u,M_u}}{\rho}\cdot d_{i,f}. \tag{7.32}$$

Finally, to deal with shared layers, the constraint defined in Eq. (7.12) is introduced per each shared layer, whereas the constraints in Eqs. (7.4), (7.5), and (7.6) are modified accordingly, as detailed in Section 7.2 with Eqs. (7.13), (7.14), and (7.15).

As an example, it is provided the complete extension to the case where the first two layers of two DLMs are shared. The additional constraints required to ensure that the shared layers $j = 1$ and $j = 2$ (of DLMs $u = 1$ and $u = 2$) are assigned to the same node i are:

$$\forall i \in \mathbb{N}_N \qquad \alpha_{1,i,1} = \alpha_{2,i,1}, \tag{7.33}$$

$$\forall i \in \mathbb{N}_N \qquad \alpha_{1,i,2} = \alpha_{2,i,2}. \tag{7.34}$$

Then, the Eqs. (7.4), (7.5), and (7.6) are modified as follows:

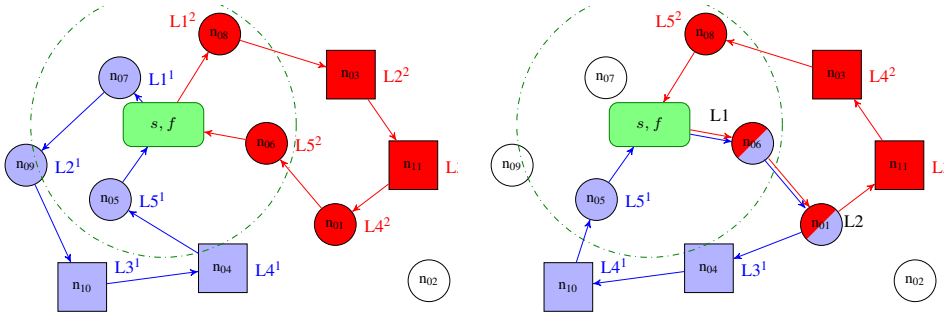
$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j} \le L + \alpha_{1,i,1} + \alpha_{1,i,2}, \tag{7.35}$$

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot m_{u,j} \le \bar{m}_i + \alpha_{1,i,1}\cdot m_{1,1} + \alpha_{1,i,2}\cdot m_{1,2}, \tag{7.36}$$

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C}\sum_{j=1}^{M}\alpha_{u,i,j}\cdot c_{u,j} \le \bar{c}_i + \alpha_{1,i,1}\cdot c_{1,1} + \alpha_{1,i,2}\cdot c_{1,2}. \tag{7.37}$$

It is noteworthy to point out that there is no difference in defining Eqs. (7.35), (7.36), and (7.37) with the variables of the DLM $u = 2$, instead of those of DLM $u = 1$ as done here. It is indeed sufficient to relax the constraints with $k - 1$ variables out of k, where k represents the number of DLMs a layer is shared among, in order to count that shared layer only once.

The proposed methodology has been applied to two instances of the 5-layer DLM described in Figure 7.1a without common processing layers operating in the IoT system



(a) *The methodology outcome, with no shared layer between the two 5-layer DLMs.*

(b) *The methodology outcome, with the first two layers shared between the two 5-layer DLMs.*

Figure 7.3: *Distributing two (i.e., $C = 2$) 5-layer DLMs (shown in Figure 7.1a), over the IoT system in Figure 7.1b, with $L = 1$.*

depicted in Figure 7.1b and with $L = 1$. Interestingly, the outcome of the methodology, depicted in Figure 7.3a, shows that the placement of both DLMs represents the optimal solution of the single-DLM configuration.

Moreover, the methodology has been applied to the case where the convolutional layers $L1$ and $L2$ are shared between the two DLMs. This solution is inspired by the transfer learning paradigm, where two DLMs might share low-level representation processing layers, while high-level ones are specific for each DLM. The methodology outcome in this scenario is interesting, showing that common layers $L1$ and $L2$ have been placed in IoT units n_{06} and n_{01}, respectively, while, after n_{01}, the processing takes two different paths.

7.6 Open Points and Possible Extensions

Currently, the proposed methodology neither takes into account the energy status of IoT units nor network failures (Farhan et al., 2017). The network failures could be managed by modifying the transmission time defined in Eq. (7.11) as

$$t_t = \left(1 + \xi^{i,k}\right) \cdot \frac{K_{u,j}}{\rho} \cdot d_{i,k}, \tag{7.38}$$

where $\xi^{i,k}$ represents the probability that a failure happens (and thus a retransmission is required) for the pair of nodes (i, k).

It is noteworthy to point out that, thanks to the transfer learning paradigm, the hierarchy of layers of the DLMs can be considered as general feature extractors (Yosinski et al., 2014). For this reason, the deployed DLMs can be easily reconfigured to address a different problem by replacing only upper layers. Moreover, this optimization phase can be scheduled periodically or when needed to manage variations in the IoT network configuration (e.g., due to the removal or insertion of IoT units). This is a crucial aspect in the scenario of mobile IoT units, a case that is not considered in this paper. In fact, thanks to the transfer learning approach and by periodically recomputing the DLM allocation, the methodology could be applied to units changing their position in the environments they are operating in.

The remaining of the section suggests two ways of dealing with the energy in the proposed methodology.

7.6.1 Introducing the Energy in the Proposed Methodology

Sections 4.3 and 7.1 formalize the system of possibly heterogeneous IoT units and the DLMs (either with or without Early-Exits) to be mapped onto it. In this Section, such a formalization is extended by introducing new variables to take into account the energy technological constraints the IoT units might introduce, i.e., removing the (implicitly) assumption made up to now of having the IoT units connected to a power source.

Let $\bar{e}_i$, for each $i \in \mathbb{N}_N$, be the energy budget for an IoT unit i, i.e., the amount of energy that causes the IoT unit either to run out of energy or to enter in power saving mode and stop to process the assigned computation. Let $e_{u,j}$, for each $u \in \mathbb{N}_C$ and for each $j \in \mathbb{N}_M$, be the energy required by the layer j of DLM u, computed as in Eq. (2.5). Let also ω_u, for each $u \in \mathbb{N}_C$, be the number of times the DLM u has to be processed before the IoT unit is deployed on runs out of energy.

Given these additional variables, a few different models to handle the energy are proposed in the following. Other possible solutions or cases, as the possibility for the nodes to harvest energy –as in (Bozorgchenani et al., 2020)– are left for future work.

Penalize the Low-Energy Nodes. The first approach focuses on the IoT units only. At first, all the IoT units i with a low-energy budget $\bar{e}_i$, by forcing the corresponding variables $\alpha_{\cdot,i,\cdot}$ to be zero, i.e., by introducing the following additional constraints to the optimization model:

$$\forall i \in \mathbb{N}_N \text{ s.t. } \bar{e}_i \leq \tilde{E}, \qquad \sum_{u=1}^{C} \sum_{j=1}^{M} \alpha_{u,i,j} = 0, \tag{7.39}$$

where $\tilde{E}$ is a further parameter modeling the amount of energy budget $\bar{e}_i$under which an IoT unit is excluded by the computation. However, an IoT unit with a low energy budget is not "removed" from the IoT system the optimization problem views. As a consequence, such an IoT unit can still be considered as an intermediate hop during the transferring of activations among IoT units processing consecutive layers of the DLM.

As a final remark, this formalization does not guarantee that the DLMs distributed onto the IoT system can be executed at least one time unless the value of $\tilde{E}$ is appropriately defined. The simplest way to achieve such a goal is to have $\tilde{E}$ greater to the maximum energy requirement introduced by any of the considered DLMs multiplied by the number of layers the methodology can place on each IoT unit, i.e.,

$$\tilde{E} \geq L \cdot \max_{u \in \mathbb{N}_C, j \in \mathbb{N}_M} e_{u,j}. \tag{7.40}$$

However, in real application scenarios, the value of $\tilde{E}$ should be a balance between the number of IoT units that are cut off and the number of times each DLM should be processed. In the following, the energy problem introduces the DLMs into its formalization.

Grant One DLM Pipeline Processing. The second energy management approach aims to guarantee that the DLMs processing pipelines are executed at least once. To achieve

this goal, similarly to Eqs. (7.5) and (7.6) for memory and computation, respectively, the following constraints are added to the optimization problem in order to ensure that the IoT units have enough energy to carry out the processing they are assigned to. Formally, the energy constraints are:

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C} \sum_{j=1}^{M} \alpha_{u,i,j} \cdot e_{u,j} \leq \bar{e}_i. \tag{7.41}$$

Grant Many DLM Pipeline Processing. The final approach presented in this work slightly extends the previous approach by guaranteeing that each employed DLM u can be processed at least ω_u times. The energy constraints in Eq. (7.41) are redefined as follows:

$$\forall i \in \mathbb{N}_N \qquad \sum_{u=1}^{C} \sum_{j=1}^{M} \omega_u \cdot \alpha_{u,i,j} \cdot e_{u,j} \leq \bar{e}_i. \tag{7.42}$$

CHAPTER 8

Privacy Is All You Need in Deep-Learning-as-a-Service

In recent years, the technological evolution of Cloud-based computing infrastructures intercepted the ever-growing demand for machine and deep-learning solutions leading to the novel paradigms of *machine* and *deep-learning-as-a-service* (Yao et al., 2017). The core of such computing paradigms is that Cloud providers provide ready-to-use remotely-executable machine/deep learning services in addition to virtual computing environments (as in infrastructures-as-a-service) or platform-based solutions (as in platforms-as-a-service). Examples of such services are the identification of faces in images or videos or the conversion of text-to-speech or speech-to-text (Hossain and Muhammad, 2015). From the perspective of the user, being ready-to-use, these services do not require the training of the models (that are pre-trained by the Cloud provider) nor the local recall of such models (that are executed on the Cloud). Moreover, the Cloud-based computing infrastructure providing such machine/deep learning solutions *as-a-service* allows to support scalability, availability, maintainability, and pay-per-use billing mechanisms (Erl et al., 2013). In the context of this work, relying on such services might be an alternative (at least partially) to approximation techniques (Sections 5.1 and 5.2) or to distributed DLMs (Chapter 7) in order to enable particularly complex DLMs, at the expenses of depending on the connectivity to the Cloud providing the services.

Unfortunately, to be effective, machine and deep-learning-as-a-service approach involves the processing of data that might be sensitive, e.g., personal pictures or videos, medical diagnoses, as well as data that might reveal ethnic origin, political opinions, but

also genetic, biometric, and health data (Council of European Union, 2016).

This chapter aims to introduce a novel distributed architecture meant to preserve user data privacy in the deep-learning-as-a-service computing scenario. To achieve this goal, the proposed architecture relies on Homomorphic Encryption (HE) that is an encryption scheme allowing the process of encrypted data (Acar et al., 2018). In the proposed architecture, by exploiting the properties of HE, users can locally encrypt their data through a public key, send them to a suitably encoded Cloud-based deep-learning service (provided through the deep-learning-as-a-service approach), and receive back the encrypted results of the computation that are locally decrypted through the private key. More specifically, such architecture allows to decouple the encryption/decryption phases, which are carried out on the user's device (e.g., a personal computer or a mobile device), from the deep-learning processing, which is carried out on the Cloud-based computing infrastructure. Such a HE-based distributed architecture allows to preserve data privacy (plain data are never sent to the Cloud provider) while guaranteeing scalability, availability, and high performance provided by Cloud-based solution.

The ability to process encrypted data of HE comes at two main drawbacks. First, the computational load and the memory demand of HE-encoded operations are much higher than regular ones, hence making the HE-encoded deep-learning processing highly demanding in terms of computation and memory. This is the reason why the focus is on a deep-learning-as-a-service approach where the computation is carried out on high-performing units on the Cloud. Second, HE supports only a limited set of operations (typically sums and multiplications). For this reason, prior to the encoding provided by the HE scheme, the deep-learning models have to be redesigned and retrained taking into account the constraints on the set of available operations. In addition, HE schemes have to be configured through some parameters that trade off the accuracy in the computation with the computational loads and memory occupation. Such a configuration, which depends on the processing chain and the data to be processed, is managed at the Cloud-level by providing different settings of parameters that the user can explore.

The proposed architecture, presented in (Disabato et al., 2020), is intended to work with any machine and deep learning solution. However, in this work, it has been tailored to image analysis solutions leveraging Convolutional Neural Networks (CNNs), and implemented through a client (locally executed on the user device) developed as a Python library and a server developed as a deep-learning-as-a-service container implemented on Amazon AWS. The developed architecture relies on a REpresentational State Transfer (REST) paradigm for exchanging encrypted data and results between client and server, while messages rely on JSON format.

8.1 Homomorphic Encryption Background

The homomorphic scheme encryption is a special type of encryption that allows (a set of) operations to be performed on encrypted data, i.e., directly on the ciphertexts. More specifically, as detailed in (Boemer et al., 2019b), an encryption function E and its decryption function D are homomorphic w.r.t. a class of functions $\mathcal{F}$ if, for any function $f \in \mathcal{F}$, we can construct a function g such that $f(x) = D(g(E(x)))$, for a set of input x.

The considered HE scheme is the Brakerski/Fan-Vercauteren (BFV) scheme (Fan and Vercauteren, 2012) that relies on the Ring-Learning With Errors (RLWE) problem, sim-

ilarly to other works (Cheon et al., 2017; Brakerski et al., 2014). (Lyubashevsky et al., 2010) provides a detailed description of the problem and its security and implementation aspects. However, this section briefly introduces the main concepts. The BFV scheme relies on the following set of encryption parameters (from now on denoted with Θ):

- m: Polynomial modulus degree;
- p: Plaintext modulus;
- q: Ciphertext coefficient modulus.

The parameter m must be a positive power of 2 and represents the degree of the cyclotomic polynomial $\Phi_m(x)$. The plaintext modulus p is a positive integer that represents the module of the coefficients of the polynomial ring $R_p = \mathbb{Z}_p[x]/\Phi_m(x)$ (onto which the RLWE problem is based). Finally, the parameter q is a large positive integer resulting from the product of distinct prime numbers and represents the modulo of the coefficients of the polynomial ring in the ciphertext space. A crucial concept of a HE scheme is the Noise Budget (NB), an indicator related to the number of operations that can be done on a ciphertext while guaranteeing the correctness of the result. This problem (i.e., the maximum number of operations on the ciphertext) comes from the fact that, during the encryption phase, noise is added to the ciphertexts to guarantee that, being $p_1 = p_2$ two plain values to be encrypted with the same public key, the corresponding ciphertexts c_1 and c_2 are different (i.e., $c_1 \neq c_2$). All the operations performed on the ciphertext consume a certain amount of NB (depending on the type of operation and the input): operations like additions and multiplications between ciphertext and plaintext consume a small amount of NB, while multiplications between ciphertexts are particularly demanding in terms of NB. When the NB decreases to 0, decrypting that ciphertext will produce an incorrect result.

From a practical point of view, the choice of the encryption parameters Θ determines several aspects: the initial value of the NB, its consumption during computations (hence the number of operations to be performed on a ciphertext), the level of security against ciphertext attacks, the computational load and memory occupation of the HE processing, and the accuracy of the results (i.e., measuring the correctness of the decrypted values). For example, the initial NB increases with m at the expense of larger memory occupation and computational loads. The plaintext modulus p is directly related to the accuracy of the HE processing. Despite being a complicated parameter to be tuned, the theory states that larger values of p will produce more accurate results at the expense of more considerable reductions of the NB. Finally, the parameter q influences both the initial NB and the level of security of the encryption. A detailed description of the parameters and their effect on the HE scheme can be found in (Laine, 2017).

It is crucial to stress that choosing the best parameter configuration is a trade-off between accuracy and performance and depends on the type and complexity of the processing, the set of feasible operations, and the available computational resources. Practical guidelines to choose Θ will be given in Section 8.3.4.

8.2 Homomorphic Encryption Related Literature

The idea of using HE to preserve the privacy of data during the computation has been introduced in (Rivest et al., 1978). In this work, *privacy homomorphisms* are defined as encryp-

tion functions that allow one to operate on encrypted data without preliminarily decrypting the operands (Rivest et al., 1978). The first HE schemes allow only additions (Naccache and Stern, 1998; Okamoto and Uchiyama, 1998; Paillier, 1999), or multiplications (ElGamal, 1985).

The first homomorphic encryption scheme allowing both multiplication and additions has been proposed in (Gentry, 2009). There, the idea was to rely on ideal lattice-based cryptography to provide a scheme supporting additions and multiplications with theoretically grounded security guarantees. After that, (Van Dijk et al., 2010) extended this work by relaxing the ideal lattice assumption (and its security) but allowing the usage of integer polynomial rings to define the cyphertexts. (Brakerski et al., 2014) introduces the Brakerski-Gentry-Vaikuntanathan (BGV) scheme that relies on polynomial rings to define the cyphertexts and on the learning with error (LWE) and ring learning with errors (RLWE) problems to provide theoretically grounded security guarantees. The RLWE problem is also the basis of the Brakerski/Fan-Vercauteren (BFV) scheme (Fan and Vercauteren, 2012), detailed in Section 8.1, and the Cheon-Kim-Kim-Song (CKKS) scheme (Cheon et al., 2017), which extends the polynomial rings to the complex numbers and isometric rings.

The HE schemes mentioned above are theoretical and, to be applied, have then been implemented to specific processing chains. As regards deep learning solutions, CryptoNets (Gilad-Bachrach et al., 2016) relies on the HE BFV scheme to execute CNNs on encrypted inputs by introducing several possible ways of approximating the non-linear computation characterizing many layers of a CNN. Similarly, (Bourse et al., 2018) provides a fast HE scheme for the (discretized) CNN inference. Recently, the nGraph-HE framework (Boemer et al., 2019a) has been proposed. This framework allows to train CNNs in plaintext on a given hardware and deploy trained models to HE cryptosystems operating on encrypted data. Unfortunately, these works are specific to a given DL solution (e.g., CNNs in (Gilad-Bachrach et al., 2016)). In contrast, our architecture is meant to be general-purpose and able to hide the complexity of adopting HE solutions, similarly to what was proposed in (Boemer et al., 2019a), still maintaining the *as-a-service* paradigm.

The literature also presents works aiming at offering encrypted computation. For example, (Yao, 1982) proposed the Secure Multi-Party Computation (SMC) approach, where more than one actor (namely, a party) collaborates in computing a function and having only partial knowledge of the data they are working on. These solutions do not encompass HE. (Barni et al., 2006) applied SMC with the Pailler HE (Paillier, 1999) to CNNs, where a party owns the data and another owns the CNN. Hence, both the data and CNN are kept secret during the computation. Other examples can be found in (Mohassel and Zhang, 2017; Rouhani et al., 2018). Finally, the Gazelle framework (Juvekar et al., 2018) relies on SMC and HE to provide low-latency inference for CNN.

8.3 The Proposed HE-DL Privacy-Preserving Cloud-Based Architecture

Figure 8.1 shows the proposed privacy-preserving distributed architecture for deep-learning-as-a-service, called *HE-DL*. More specifically, *HE-DL* relies on a distributed approach where the user device, given a pair of keys (public k_p and secret k_s) and the HE parameters Θ, carries out the *Encryption* $E(x, \Theta, k_p)$ of user data x and the *Decryption*

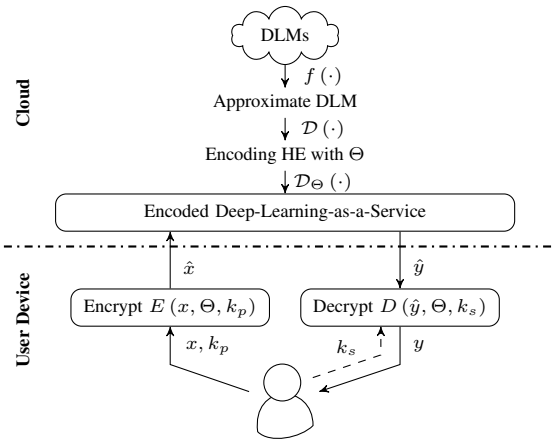


Figure 8.1: *The proposed privacy-preserving architecture for deep-learning-as-a-service.*

$D(\hat{y}, \Theta, k_s)$ of processed data $\mathcal{D}_\Theta(\hat{I})$. Both $E(\cdot)$ and $D(\cdot)$ are based on the HE-BFV scheme described in Section 8.1.

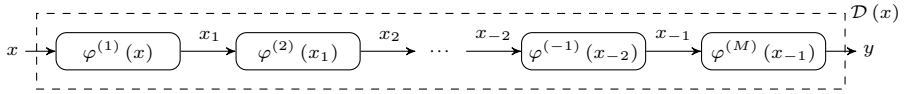
Conversely, the Cloud carries out the (encrypted) deep learning processing $\mathcal{D}_\Theta(\cdot)$. This is a crucial step since deep learning processing is typically highly demanding in terms of computational load and memory occupation. It is crucial to point out that, as commented in Section 8.1, the considered deep-learning-as-a-service computation has to be approximated by using only addition and multiplication in order to process the ciphertext $\hat{x}$. For this reason, the set of *DL models* $f(\cdot)$s that are made available by the Cloud are approximated through addition and multiplication, i.e., defining the set of approximated *DL models* $\mathcal{D}(\cdot)$s. Once approximated, $\mathcal{D}(\cdot)$s have to be encoded following the rule of the HE-BFV scheme to get the *encoded deep-learning-as-a-service* $\mathcal{D}_\Theta(\cdot)$ by relying on the HE parameters Θ. This encoding phase converts the plain values parameters of the *DL models* in a form that allows the computation through the HE-BFV scheme on encrypted inputs $\hat{x}$s.

The *DL models* considered in this work are the CNNs aiming at classifying the input images x into a class $y \in \Delta$. In such a scenario, the proposed *HE-DL* makes available the deep-learning-as-a-service computing paradigm into two different modalities:

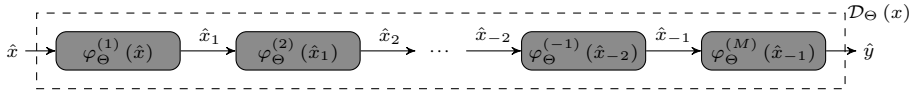
- *recall*: the processing $\mathcal{D}_\Theta(\cdot)$ provides the encrypted version $\hat{y}$ of the final classification y of x;

- *transfer learning*: the processing $\mathcal{D}_\Theta(\cdot)$ provides the encrypted version of a processing stage of the considered CNN applied to the input image x. The final classification y is carried out on the user device thanks to a suitably-trained classifier (e.g., a Support Vector Machine or a neural based classifier).

These two modalities will be detailed in the rest of the section, together with the description of the encryption and decryption phases, the approximation and encoding of CNNs, the configuration of the encryption parameters, and the communication between the user device and Cloud. Please refer to (Disabato et al., 2020) for details about the HE-DL im-



(a) *The plain processing of an approximated CNN $\mathcal{D}(\cdot)$ with M layers. Each layer φ^{ℓ}, with $1 \leq \ell \leq M$ has here only multiplications and/or additions (negative values for ℓ are shortcuts for $M - \ell$). The difference w.r.t. the* classic *CNN-based classifier $f(\cdot)$ relies on such constraint. The layers parameters θ_ℓ, may be approximated (and referred to as $\hat{\theta}_l$) in $\mathcal{D}$.*



(b) *The encrypted processing of the CNN $\mathcal{D}_\Theta(\cdot)$. The CNN is encoded with HE parameters Θ, processes inputs $\hat{x}$s encrypted with Θ as well, and returns the encrypted classification output $\hat{y}$.*

Figure 8.2: *A comparison of the plain and approximated CNN processing with the encrypted one. The layers' parameters θ_ℓs are omitted to simplify the notation.*

plementation. Finally, Section 10.2 experimentally evaluates the HE-DL architecture in both the modalities.

8.3.1 Encryption and Decryption

The *encryption* function $E(x, \Theta, k_p)$ transforms (based on the HE-BFV scheme) a plain input x into an encrypted one $\hat{x}$ given the HE encryption parameters Θ with the support of a public key k_p. The *decryption* function $D(\hat{y}, \Theta, k_s)$ operates on the encrypted output $\hat{y}$ of the computation $\mathcal{D}_\Theta(\hat{x})$, being $\hat{x}$ the encrypted input. More specifically, $D(\hat{y}, \Theta, k_s)$ computes the plain output y given the same set of parameters Θ and the secret key k_s (corresponding to k_p). The semantic of y depends on the considered working modality of *HE-DL*:

- y is the classification label of the input x in the *recall* modality;
- y is an array of extracted features representing the values of the activation function of a given layer of the CNN in the *transfer learning* modality.

8.3.2 Approximated and Encoded DL Processing

The proposed architecture *HE-DL* is general enough to employ a wide range of machine and deep learning models. In this chapter, the focus is on CNNs for two main reasons. First, CNNs are widely-used and effective solutions for image classifications. Second, for most of their processing, CNNs are composed of addition and multiplication operations, making them suitable candidates to be considered within a HE scheme.

Let $f(\cdot)$ be a CNN of M layers $\eta_{\theta_\ell}^{(\ell)}$ with parameters θ_ℓ and $\ell = 1, \ldots, L$, aimed at extracting features and providing the classification output y of an input image x.[15]

As mentioned above, in order to be used with HE, CNNs have to be approximated by considering only computing layers and activation functions that are suitable for the

[15]The formalization of a CNN follows that of a Deep Learning Model in Section 4.2, being the CNN a DLM.

considered HE-BFV scheme. Given that only addition and multiplication are allowed, only polynomials functions can be computed directly, while non-polynomials operations must be either approximated with a polynomial form or replaced with other (allowed) operations. Similarly to what done in (Gilad-Bachrach et al., 2016), in the proposed *HE-DL* architecture, the following rules allow to define the approximated CNN model $\mathcal{D}(\cdot)$ from the original CNN $f(\cdot)$:

- the ReLU activation function is replaced with a Square activation function that simply squares the input value;
- the max-pooling operator is replaced with the average one, with the division converted to a multiplication by $\frac{1}{f_s}$, being f_s the pooling size (fixed and a-priori known).
- Rules to approximate other non-polynomial layers can be found in (Gilad-Bachrach et al., 2016).

The result of this approximation is a CNN $\varphi(\cdot)$ whose processing layers $\varphi^{(\ell)}_{\hat{\theta}_\ell}$ can be encoded with the considered HE-BFV scheme (Figure 8.2a). To simplify the notation, the parameters of each layer θ_l (or $\hat{\theta}_l$ when encoded) are omitted from now on. It is essential to point out that, after replacing the non-polynomial layers, the model has to be trained again. This is necessary because the weights of the plain model are not valid anymore if the activation functions or other layers have been replaced by different ones. Hence, to provide a deep-learning-as-a-service, $\mathcal{D}(\cdot)$ must be retrained with the same settings in which the plain one was trained (e.g., same dataset and same learning algorithm). Obviously, if the original model $f(\cdot)$ already contains HE-compatible processing layers, this procedure is not necessary. Moreover, this approximation process can introduce a variation in the accuracy between $f(\cdot)$ and $\mathcal{D}(\cdot)$. This aspect will be explored in Section 10.2.

In order to work with the encrypted inputs $\hat{x}$s, the suitably approximated CNN $\mathcal{D}$ must be encoded with the parameters Θ as defined by the HE-BFV scheme leading to the encoded CNN $\mathcal{D}_\Theta(\cdot)$ (Figure 8.2b).

Summing up, the HE-based *encrypted processing* (Figures 8.1 and 8.2) can be formalized as follows:

$$y = D(\hat{y}, \Theta, k_s) = D(\mathcal{D}_\Theta(\hat{x}), \Theta, k_s) = D(\mathcal{D}_\Theta(E(x, \Theta, k_p)), \Theta, k_s), \tag{8.1}$$

where $\hat{y}$ represents the input x's encrypted classification.

8.3.3 The Two Modalities: Recall and Transfer Learning

As mentioned above, the deep-learning-as-a-service computing paradigm has two different modalities, *recall* and *transfer learning*. The difference between them lies in how Eq. (8.1) is implemented. The former operates on the decrypted output y of the CNN $\mathcal{D}$ last layer M (typically a softmax on top of a classification layer), whereas the latter one works on the features $x_{\tilde{\ell}}$ extracted at a given CNN level $\tilde{\ell}$, with $1 \leq \tilde{\ell} < M$ (typically a convolutional or pooling one). The two modalities are detailed in what follows.

Recall. In recall, the user relies on one of the ready-to-use encoded CNN $\mathcal{D}_\Theta(\cdot)$s to classify the image x. More precisely, the user wants the image x to be encrypted into

$\hat{x}$ and to be forwarded through all the layers of the encoded CNN $\mathcal{D}_\Theta$, obtaining the final encrypted result $\hat{y}$ of the classification task, without transmitting the image x to the service provider. The assumption underlying this modality is that the chosen model $\mathcal{D}_\Theta(\cdot)$ has been trained to classify images of the same domain of the input image x.

Transfer Learning. When the application problem of the user is not matched by any model $\mathcal{D}_\Theta(\cdot)$s (e.g., the user wants to distinguish between cars and bikes while available models have been trained to classify digits or faces), the transfer learning modality comes into play. In fact, following the transfer learning paradigm (Yosinski et al., 2014; Alippi et al., 2018), the processing of a pre-trained CNN can be split into two parts: feature extraction and classification. The feature extraction processing represents a pre-trained feature extractor able to feed an ad-hoc classifier trained on the specific image classification problem (that can be different from the one originally used to train the CNN). This allows to use part of a pre-trained CNN and train only a final classifier (hence reducing both the training complexity and the number of required images).

In our scenario, the encrypted input images $\hat{x}$s will be forwarded through the encoded model $\mathcal{D}_\Theta$ up to a layer $\tilde{\ell}$. More specifically, $\mathcal{D}_\Theta$ comprises layers from 1 to $\tilde{\ell}$, with $1 \leq \tilde{\ell} \leq L$, whereas all the (possibly) remaining layers, from $\tilde{\ell}+1$ to the final one L remain plain and (might) operate on the decrypted output of layer $\tilde{\ell}$, i.e.,

$$x_{\tilde{\ell}} = D\left(\mathcal{D}_\Theta^{\tilde{\ell}}\left(E\left(x, \Theta, k_p\right)\right), \Theta, k_s\right), \tag{8.2}$$

where $\mathcal{D}_\Theta^{\tilde{\ell}}$ represents the encoded CNN $\mathcal{D}$ up to layer $\tilde{\ell}$ with parameters Θ. In this scenario, the features extracted from the encrypted input $\hat{x}$ are the output of the model $\mathcal{D}_\Theta^{\tilde{\ell}}$.

It is noteworthy to point out that following such an approach, the user can locally train a classifier on the decrypted vectors $x_{\tilde{\ell}}$ as follows. A set of K images $\left\{x^1, \ldots, x^K\right\}$ is sent to *HE-DL* that provides the corresponding outputs $\left\{\hat{x}_{\tilde{\ell}}^1, \ldots, \hat{x}_{\tilde{\ell}}^K\right\}$, that are locally decrypted into $\left\{x_{\tilde{\ell}}^1, \ldots, x_{\tilde{\ell}}^K\right\}$. The combination of this set of decrypted features with the corresponding labels (that are available to the user) defines a training set to locally train a classifier. Once trained, the system is ready-to-use: the user can send an encrypted image $\hat{x}$ to the Cloud, receive the CNN output $\hat{x}_{\tilde{\ell}}$, decrypt it to $x_{\tilde{\ell}}$, and apply the classifier on $x_{\tilde{\ell}}$ to obtain the predicted classification label $\hat{y}$ of x.

8.3.4 Encryption Parameters

As already mentioned, the choice of the HE parameters $\Theta = \{m, p, q\}$ is critical to get correct processing of the encrypted input $\hat{x}$. The choice for q is tough and influences the security of the scheme. For this purpose, SEAL library (SEAL, 2019) provides a specific function that, given the polynomial modulus degree m and the desired AES-equivalent security level (sec), returns a suggested value for q (Laine, 2017). In this work, sec has been set to 128 *bits*, which is the default value of SEAL. The value of q has been set accordingly. Then, the other parameters possibly values are $m \in \{1024, 2048, 4096\}$ and $p \in \{32, 712, 37780, 1.3 \cdot 10^5, 1.5 \cdot 10^5, 2.6 \cdot 10^5, 5.2 \cdot 10^5, 6.0 \cdot 10^5, 2.1 \cdot 10^6, 1.3 \cdot 10^8, 1.5 \cdot 10^8\}$, with the goal of having an initial NB big enough to carry out the required CNN computation.

8.3.5 Communication Between User Device and Cloud

The communication between the User Device and the Cloud is carried out through a JSON-format message. More specifically, being an on-demand computation, clients have to perform a request to the online deep-learning-as-service provider, including:

- a set of parameters, including the encryption parameters m and p, the security level sec, the identifier of the chosen DLM $\mathcal{D}_\Theta(\cdot)$, and the specific layers to end the computation at (which will determine the modality, i.e., recall or transfer learning);
- the encrypted input $\hat{x}$ on which the computation is performed, which has to be encrypted using a public key generated according to the encryption parameters.

The provider publishes information about the available models. Once the computation has been carried out, the Cloud responds with a JSON message containing the encrypted result vector $\hat{y}$ (or $\hat{x}_{\tilde{\ell}}$ in transfer learning mode).

Part IV

Experimental Results

CHAPTER 9

Evaluating Deep Tiny Machine Learning Solutions

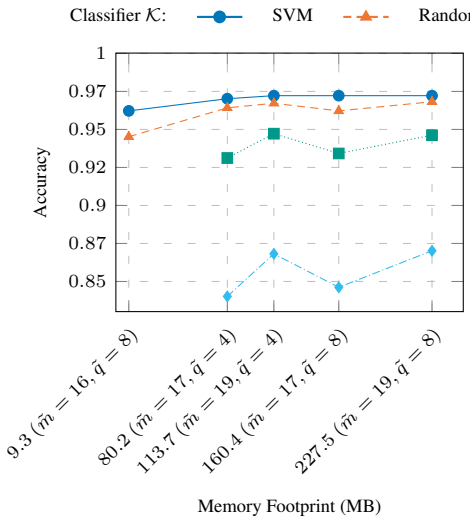
This chapter aims to collect the results about proposed solutions for Deep Tiny Machine Learning in Part II.

9.1 Evaluating the Methodology to design Deep Tiny Machine Learning

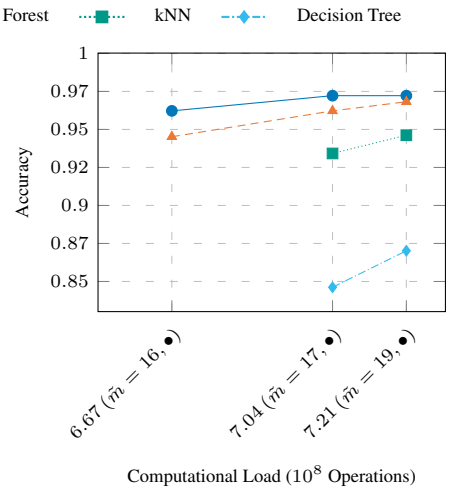
The methodology to design Deep Tiny Machine Learning solutions, presented in (Alippi et al., 2018) and detailed in Section 5.1, has been evaluated in two steps. At first, Section 9.1.1 tests the methodology synthetically on different configurations of the number of layers $\tilde{m}$, the number of fractional digits as configurations $\tilde{q}$ in precision scaling, and the kind of classification algorithm $\mathcal{K}$. After that, Section 9.1.2 evaluates the methodology on two off-the-shelf embedded platforms: the STM32F7 MCU, with a 167 MHz Cortex-M7 core, 512 KB of RAM, and no operating system; and the Raspberry Pi 3B, with a 1.2 GHz ARM-11 core and 512 MB of RAM.

9.1.1 Synthetic analysis on image recognition

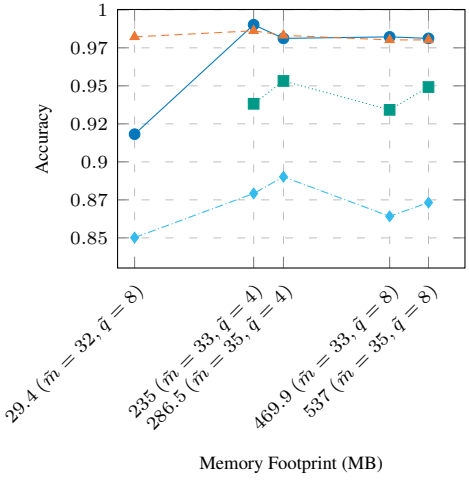
The synthetic evaluation considers different configurations of $\tilde{m}$, $\tilde{q}$, $\mathcal{D}$, all with $\delta = 3$ classes to classify among. The DLMs are the AlexNet (Krizhevsky et al., 2012) and the VGG-16 (Simonyan and Zisserman, 2014). Instead, the classification algorithms $\mathcal{K}$ are an



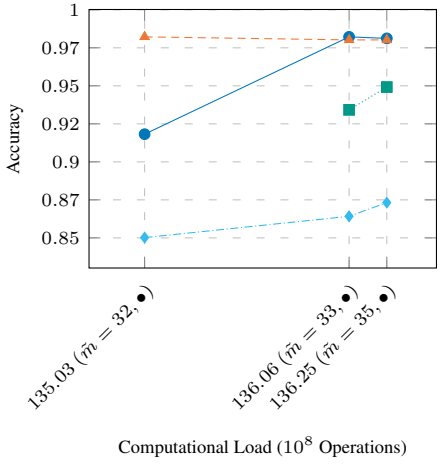
(a) *The memory footprint of the models $\hat{\mathcal{D}}$ derived from the AlexNet with different number of layers $\tilde{m}$ and fractional bits $\tilde{q}$.*



(b) *The computational load of the models $\hat{\mathcal{D}}$ derived from the AlexNet with different number of layers $\tilde{m}$*



(c) *The memory footprint of the models $\hat{\mathcal{D}}$ derived from the VGG-16 with different number of layers $\tilde{m}$ and fractional bits $\tilde{q}$.*



(d) *The computational load of the models $\hat{\mathcal{D}}$ derived from the VGG-16 with different number of layers $\tilde{m}$.*

Figure 9.1: *The results of the methodology to design Deep TML solutions for embedded systems or IoT units, with $\delta = 3$ classes, different DLMs, and classifiers $\mathcal{K}$.*

SVM, a Random Forest with 100 Decision Trees, a k-Nearest Neighbors (with k equal to the ceiling of the square root of the training set number of samples), and a single Decision Tree. All the experiments rely on the *Caltech 256* benchmark, which has 256 classes with

Table 9.1: *Porting an AlexNet-based image-recognition application to an STM32F7 MCU and a Raspberry Pi 3B, with $\tilde{m} = 5$, i.e., the first convolutional layer along with the local response normalization, and the max-pooling.*

	STM32F7	**Raspberry Pi 3B**		
CPU	ARM M7@167 MHz	ARM11@1.2 GHz		
RAM	512 KB	1024 MB		
$\bar{m}$	102 KB	102 KB		
$\bar{c}$	$100 \cdot 10^6$	$100 \cdot 10^6$		
$\tilde{m}$	5	5	5	5
$\tilde{q}$	8	8	8	4
Filter-Selection?	yes	no	no	no
No. Filters	1	1	-	-
$\mathcal{K}$	Decision Tree	Decision Tree	SVM	SVM
$\hat{a}$	87.9	87.9	99.3	99.4
$m_{\hat{\mathcal{D}}}$	1.4 KB	1.4 KB	68 KB	34 KB
$c_{\hat{\mathcal{D}}}$	$1.09 \cdot 10^6$	$1.09 \cdot 10^6$	$52.7 \cdot 10^6$	$52.7 \cdot 10^6$
Inference Time (ms)	2700	178	8687	8687

at least 80 images per class, and it is not the same dataset used in the DLMs training. The figure of merit is the accuracy of the resulting approximated Deep TML algorithms $\hat{\mathcal{D}}$s, computed as an average of 100 experiments with 70-30 as training-test split and the number of samples per class equal to the minimum between two-hundred and the minimum number of samples in the considered classes.

Figure 9.1 shows the experimental results. The first observation is that task dropping and precision scaling have minimal effects on classification accuracy. This allows to support the idea of the Deep TML model $\hat{\mathcal{D}}$ as an application-specific and approximated version of $\mathcal{D}$. More in details, Figures 9.1a and 9.1b compare the accuracy w.r.t. the memory footprint and the computational load for different Deep TML solutions $\hat{\mathcal{D}}$ derived from the AlexNet, respectively, whereas Figures 9.1c and 9.1d those derived from the VGG-16. Interestingly, the methodology is able to significantly reduce the memory footprint as well as the computational load, with minimal drops in accuracy. For instance, with the SVM classifier and the AlexNet, the methodology provides a reduction from 227.5 to 80.2 MB of the memory footprint and from 721 to 704 millions of operations in computational load, by moving from the configuration ($\tilde{m} = 19$, $\tilde{q} = 8$) to ($\tilde{m} = 17$, $\tilde{q} = 4$). This meaningful reduction in memory footprint and computational load comes at the expenses of a negligible reduction in $\hat{a}$, i.e., less than 1%. Similarly, there is no drop in accuracy when moving from the configuration ($\tilde{m} = 35$, $\tilde{q} = 8$) to ($\tilde{m} = 33$, $\tilde{q} = 4$) of the VGG-16, with a reduction of the memory footprint from 537 to 253 MB, i.e., more than 50%.

9.1.2 Porting the Approximated AlexNet to Two Off-the-Shelf Embedded Platforms for Image Recognition

The proposed methodology has then be tested on an image-recognition embedded application whose technological targets are two well-known off-the-shelf embedded platforms: the STM32F7 microcontroller and the Raspberry Pi 3B (see Table 9.1 for their technological details).

To allow a fair evaluation of the proposed methodology, the considered image-recog-

nition application has been modeled as a two-class classification problem (i.e., $\delta = 2$) aiming at distinguishing between the class *people* and *car* of the *Caltech-256* benchmark. The resulting dataset has 209 images of *people* and 116 of *cars*. The starting DLM is the AlexNet, the constraint on memory has been set to a fifth of the STM32F7 available RAM, i.e., $\bar{m} = 102$ KB, and the computational constraint to $\bar{c} = 100 \cdot 10^6$.

Table 9.1 shows the methodology results under these settings. The methodology is capable of designing effective Deep TML models $\hat{\mathcal{D}}$ satisfying the constraints given by the technology. In particular, for the STM32F7, the selected configuration of $\hat{\mathcal{D}}$ refers to $\tilde{m} = 5$ and $\tilde{q} = 8$, that corresponds to the first convolutional block of the AlexNet, i.e., the convolution, the local response normalization, the non-linear activation, and the max-pooling. Moreover, a filter selection mechanism has been applied to further reduce the memory footprint by selecting one filter out of 96. The resulting memory footprint is $m_{\hat{\mathcal{D}}} = 1.4$ KB, with a computational load $c_{\hat{\mathcal{D}}} = 1.09 \cdot 10^6$ and an accuracy $\hat{a} = 87.9\%$, where the employed classifier is a Decision Tree. The inference time of a simple –non-optimized– C-implementation, measured with an oscilloscope, is 2.7s.

The results about the Raspberry Pi 3B show $\hat{a}$, $m_{\hat{\mathcal{D}}}$, and $c_{\hat{\mathcal{D}}}$ for three different configurations, i.e., $\tilde{q} = 8$ and filter-selection, $\tilde{q} = 8$ without filter-selection, and $\tilde{q} = 4$ without filter-selection. As expected, when the Raspberry Pi 3B operates in the same configurations as the STM32F7, the only difference is in the execution time. Interestingly, with an SVM classifier, very high accuracy can be achieved with a model with only 34 KB as memory footprint.[16] The high inference time is probably due to the considered non-optimized C implementation.

9.2 Early-Exits Evaluation

The aim of this section is to present the experimental evaluation of the Early-Exit DLMs equipped with Gate-Classifiers, presented in paper (Disabato and Roveri, 2018) and described in Section 5.2. In particular, two Gate-Classification Early-Exit DLMs have been evaluated on both their computational load and classification ability. The Section organization follows. Section 9.2.1 presents the two employed models, whereas Section 9.2.2 and 9.2.3 detail the considered datasets as well as the figures of merit adopted in the evaluation. Finally, Section 9.2.4 presents the evaluation.

9.2.1 The employed Gate-Classification DLMs

This experimental section employs two Early-Exit DLMs based on Gate-Classification, that have been also considered in other part of this document (e.g., see Chapter 7), i.e., the 6-layer EX-DLM (see Figure 7.2a and Table 7.1a for details) and the EX-Alexnet (see Figure 10.1a and Table 7.1b for details), with slightly different Gate-Classifier and final classifier to take into account the specific classification problems.

The architecture of the 6-layer EX-DLM is designed for the CIFAR-10 dataset, with the following minor architectural differences when operating on the MNIST one: the first

[16] The $\tilde{q}$ parameter mainly affects the memory occupation, while its effect on the execution time is strictly dependent on the specific implementation (and platform): in the hardware platforms when considering the use of 16-bit FP (the simplest implementation of $\tilde{q} = 4$) in the C code does not reduce the execution time since the GCC-compiler limits the usage of this data-type to storage purposes only, casting the operands to 32-bit FP during processing.

convolutional layer has 32 filters (instead of 64), whereas both the Gate-Classification layer and the final classifiers have two fully-connected layers of 1024 and 10 neurons (instead of three layers with 384, 192, and 10 neurons).

9.2.2 Datasets

The following four datasets have been considered in the evaluation:

- *MNIST* (LeCun et al., 1998) is the famous dataset of the handwritten digits composed of 70000 grey-scale 28x28 images belonging to 10 classes. In particular, there are 55000 images for training, 5000 for validation, and 10000 for testing;
- *CIFAR-10* (Krizhevsky et al., 2009) is a dataset of 60000 RGB 32x32 images belonging to 10 categories (i.e., water, land and air vehicles, and animals). In particular, there are 45000 images for training, 5000 for validation, and 10000 for testing;
- *Cats-Dogs* (Elson et al., 2007) dataset has 25000 various-size RGB images belonging to exactly two classes, as its name says. In particular, there are 15000 images for the training, 5000 for validation, and 5000 for testing;
- *Stanford Dogs* (Khosla et al., 2011) is a dataset extracted directly from ImageNet dataset (Deng et al., 2009) with approximately 19000 various-size RGB images belonging to 120 classes of dog breeds. In particular, there are 12000 images for training and 7000 images for testing (here, due to the limited number of images in the dataset, γ is learned on the training set) and balanced by forcing the maximum number of images per class to 200.

In particular, in our experimental analysis, the *MNIST* and *CIFAR-10* datasets have been considered for the 6-layer EX-DLM, whereas the *Cats-Dogs* and *Stanford Dogs* datasets for the EX-AlexNet.

9.2.3 Figures of Merit

The experimental analysis employs the following nine figures of merit:

- $\hat{a}_1$, the accuracy of Gate-Classification layer;
- $\hat{a}_M$, the accuracy of the other classifier, i.e., at the end of the whole DLM pipeline;
- $\hat{a}$, the accuracy of the Early-Exit DLM;
- $\hat{\gamma}$, the learned value of the parameter γ;
- ζ_1, ζ_M, the percentages of exits at the Gate-Classification layer and at the end of DLM, respectively;
- c_1, c_M, with an abuse of notation, the complexities of DLM up to the Gate-Classification layer and of the whole DLM, respectively;
- $c(\hat{\gamma})$, the mean computational complexity of the Early-Exit DLM according to the value $\hat{\gamma}$.

Results are averaged over five runs and the accuracies are computed as mean and standard deviation (to provide a confidence interval).

Table 9.2: *The comparison between classic DLMs and Early-Exit-based ones.*

		MNIST (LeCun et al., 1998)	CIFAR-10 (Krizhevsky et al., 2009)	Cats-Dogs (Elson et al., 2007)	Stanford Dogs (Khosla et al., 2011)
Early-Exit DLM	$\hat{a}_1$	0.976 ± 0.006	0.765 ± 0.004	0.814 ± 0.002	0.028 ± 0.000
	$\hat{a}_M$	0.986 ± 0.001	0.776 ± 0.003	0.870 ± 0.002	0.172 ± 0.003
	$\hat{a}$	0.983 ± 0.003	0.765 ± 0.004	0.852 ± 0.003	0.150 ± 0.007
	ζ_1	0.967	1.000	0.772	0.113
	ζ_2	0.033	0.000	0.228	0.887
	$\hat{\gamma}$	0.644	0.100	0.995	0.062
	c_1	7 059 968	8 768 640	353 260 192	353 267 74
	c_M	20 316 672	30 147 136	744 215 200	744 706 080
	$c(\hat{\gamma})$	7 497 439	8 768 640	442 397 934	700 473 548
Classic DLM	$\hat{a}$	0.985 ± 0.001	0.772 ± 0.002	0.868 ± 0.003	0.173 ± 0.004
	c	13 883 904	25 254 592	720 335 648	720 834 080

9.2.4 Experimental Evaluation

Table 9.2 show the experimental results. Several relevant comments arise.

First of all, as expected, the computational loads $c(\hat{\gamma})$ of Early-Exit DLMs are significantly lower than those of traditional DLMs c. This is a crucial point for the proposed solution, whose goal is to reduce the computational load of DLMs, making them feasible to be executed in computing systems characterized by constraints on computation. The reduction in computational load is quite impressive for the Gate-Classification on the MNIST (46.3%), the CIFAR (70.9%), and the Cats-Dogs (39.6%). The reduction on the Stanford Dogs is significantly smaller –only 2.7%– probably due to the peculiarity of the classification problem (120 classes and no more than 200 images per class). These gains are justified by the high percentages ζ_1 of inputs classified at the Gate-Classification layer. More specifically, in the 6-layer DLM on MNIST, only 3.3% of the images proceed up to the end. The same DLM on CIFAR is even more interesting, showing that all the images are classified at the Gate-Classification layer. The EX-AlexNet on Cats-Dogs provides a very high $\zeta_1 = 0.772$, but, on the contrary, on the Stanford Dogs $\zeta_1 = 0.113$ meaning that a high-enough posterior probability rarely supports the classification at the Gate-Classification layer.

Second, the reduction in computational load comes at a very low or even negligible reduction in the classification accuracy. This is evident by comparing the classification accuracies $\hat{a}$ provided by the Early-Exit DLMs and those of "traditional" DLMs. Interestingly, the classification accuracy provided by the 6-layer EX-DLM on the MNIST is statistically indistinguishable from the one provided by the traditional DLM (but with a meaningful reduction in the computational load).

It is also worth noting that the classification accuracy $\hat{a}_M$ provided by the final classifier is larger than the one provided by the Gate-Classification layer $\hat{a}_1$. This is reasonable since the final classifier is able to exploit the processing of the whole DLM. What is particularly interesting is that $\hat{a}_M$s are close to or slightly higher than the $\hat{a}$s of the "traditional" DLMs. This is very promising, proving that the introduction of Early-Exits (Gate-Classification layers) significantly reduces computational load without affecting the model learning.

9.3 Adaptive Deep Learning Models Results

The aim of this section is to present the experimental evaluation of the adaptation mechanism for DLMs in presence of concept drift, presented in paper (Disabato and Roveri, 2019) and described in Section 5.3. Section 9.3.1 presents the employed DLMs and datasets, whereas Sections 9.3.2 and 9.3.3 detail the considered concept drifts and the experimental settings, respectively. Finally, Section 9.3.4 comments the results.

9.3.1 The Employed DLMs and Datasets

This experimental section relies on two Convolutional Neural Networks as DLMs: the AlexNet (Krizhevsky et al., 2012) and the Cat-Dog CNN, i.e., a variant of the AlexNet where the final classification layer has been replaced with a 4096x2 fully-connected layer. The latter DLM has been trained on the Cat-Dog dataset (Elson et al., 2007) for 25 epochs with a learning rate of 0.001, a momentum of 0.9, and a 70-30 split of training-validation images.

In addition to the already mentioned Cat-Dog dataset (Elson et al., 2007) that has two classes –Cats and Dogs– of 12 500 images each, we consider the ImageNet (Dean et al., 2012), that has 1000 classes of about 1 200 images each.

9.3.2 The Employed Concept Drift

Two different concept drifts have been considered:

- *Class Change* refers to the scenario where one or more classes in Δ change its/their information content. In particular, in our experiments, the number of classes that change their information content ranges from 1 to $\delta - 1$.

- *Camera Degradation* models a problem affecting the input source that induces a reduction in brightness, saturation or contrast whose intensity ranges from 10% to 75%. Both the type of degradation and its intensity are randomly selected.

In each experiment, the concept drift is assumed to happen after $200 \cdot \delta$ inputs.

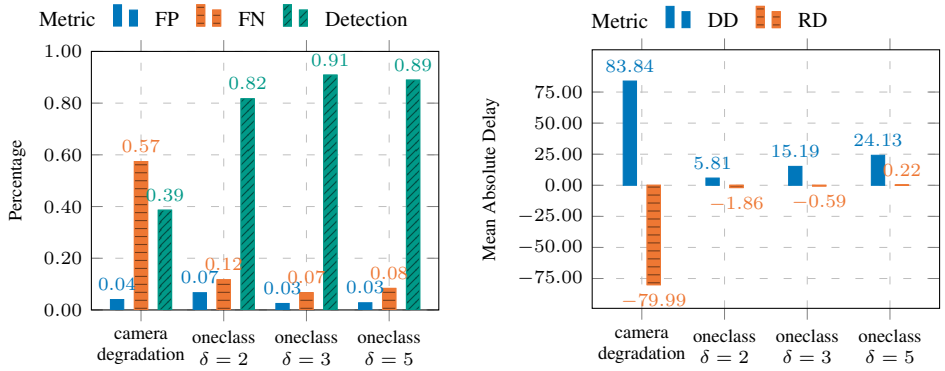
9.3.3 Experimental Settings

This experimental section compares two different solutions:

- *Pure Active DLM*, where the adaptation procedure only retrains the last classification layer $\mathcal{K}^{(M)}$ of the considered DLM,

- *Adaptive DLM*, that implements the whole solution proposed in Section 5.3.

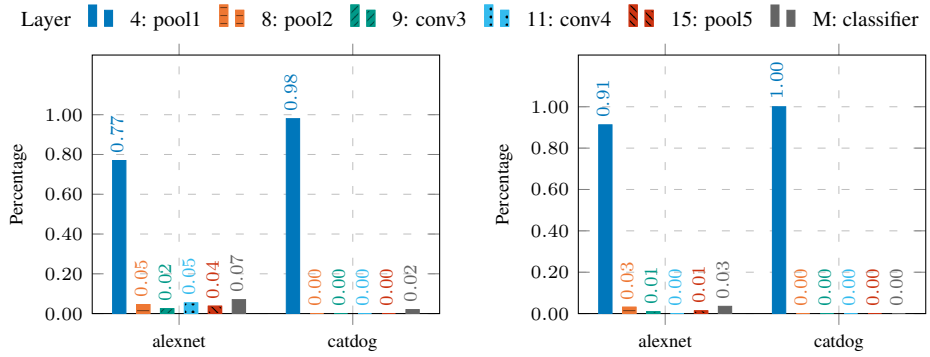
Needless to say, both the Pure Active and Adaptive DLMs share the same CDT, with threshold $h = 50$, and RP as well as the same learning algorithm used during the retraining.



(a) *The percentages of False Positives (FP), False Negatives (FN), and detections.*

(b) *The mean detection delay (DD) and refined delay (RD).*

Figure 9.2: *The experimental evaluation of the CDT and the RP on the AlexNet DLM and the ImageNet dataset.*



(a) *The results with* $\Xi = 100 \cdot \delta$.

(b) *The results with* $\Xi = 700 \cdot \delta$.

Figure 9.3: *The pipeline exploration results: the distribution of* $\tilde{\ell}$ *on the AlexNet and the Cat-Dog DLMs, with different values of* Ξ.

9.3.4 Experimental Evaluation

Evaluating the CDT and the RP. A preliminary analysis of the CDT and RP has been carried out with the AlexNet and a subset of the ImageNet dataset. The number of classes δ (randomly chosen within those of ImageNet) ranges from 2 to 5, with one of them changing when *Class Change* concept drift is employed.

Figure 9.2 shows the results of this analysis. Two main comments arise.

First, the considered CDT provides very high detection accuracy for the *Class Change* concept drift. In fact, Figure 9.2a shows that the percentage of false positive and negative detections is very low for every considered value of δ. Similarly, Figure 9.2b shows the *Detection Delay* –the difference between the change detection and the change time– and the *Refined Delay* –the difference between the estimation t_r and the real change time–

Table 9.3: *The increase in accuracy (minimum, mean, and maximum) of the Adaptive DLM, w.r.t. the Pure Active DLM, with different number of training epochs E.*

DLM	δ	$E = 10$			$E = 20$			$E = 30$		
		min	mean	max	min	mean	max	min	mean	max
AlexNet	2	0.001	0.016	0.055	0.000	0.012	0.037	0.000	0.011	0.037
	3	0.002	0.026	0.049	0.006	0.028	0.048	0.006	0.028	0.048
	5	0.004	0.021	0.040	0.003	0.039	0.090	0.003	0.040	0.092

whose values are low for *Class Change* concept drift. As expected, the detection delay increases with δ.

Second, the CDT is not effective in detecting camera degradation, as shown by the high numbers of false-negative detections in Figure 9.2a and by the high detection and refined delays in Figure 9.2b. This is probably justified by the fact that AlexNet is robust to such kind of changes.

Evaluating the Adaptation Module. The second part of this experimental analysis aims at evaluating the proposed adaptation module. The settings employ two different values of Ξ, i.e., 100δ and 700δ and the following set of convenient layers

$$\mathcal{L} = \{4 : \textit{pool1}, 8 : \textit{pool2}, 9 : \textit{conv3}, 11 : \textit{conv4}, 15 : \textit{pool5}\}$$

within both the considered DLMs.

To ease the comparison, each detection is assumed to be correct (i.e., there are no false positive or negative detections) and that its refined delay is always equal to 0.

Figure 9.3 shows the distribution of the first layer $\tilde{\ell}$ for which the null-hypothesis is rejected. Two comments arise. First, when the DLM is tailored to a specific application, as in the case of Cat-Dog CNN, the distribution of $\tilde{\ell}$ within $\mathcal{L}$ is significantly shifted towards the first layer (in this case, the layer 4, namely *pool1*). This behavior suggests that the features are application-specific. Thus the adaptation of the whole pipeline is required in almost all cases. Second, there are small difference between the results with $\Xi = 100\delta$ (Figure 9.3a) and with $\Xi = 700\delta$ (Figure 9.3b), meaning that the pipeline exploration is effective even with a reduced amount of input images.

Finally, Table 9.3 analyses the accuracy improvements of the proposed *Adaptive DLM* with respect to the *Pure Active DLM* adapting only the final classification layer, with different number of training epochs $E = \{10, 20, 30\}$. In all the configurations, the values are non-negative, meaning that the *Adaptive DLM* overcomes the *Pure Active DLM*. Moreover, these results corroborate the idea of adapting the pipeline of the DLM instead of the final classification layer only.

9.4 On-Device Tiny Machine Learning Results

The proposed on-device deep TML solutions in Chapter 6 and papers (Disabato and Roveri, 2020; 2021) have been validated on two different application scenarios (described in Section 9.4.1), two types of concept drift (defined in Section 9.4.2) and three different Micro-Controller Units (MCUs) from STMicroelectronics (whose technological details

are given in Section 9.4.6). In addition, the rest of the section is organized as follows. Section 9.4.3 discusses the experimental settings, whereas Sections 9.4.4 and 9.4.5 provides the experimental results. Finally, Section 9.4.6 presents the porting of the Hybrid Tiny kNN algorithm on the three considered MCUs.

It is noteworthy to highlight that TML in presence of concept drift is an entirely new research area. To the best of our knowledge, this is the first work in the related literature proposing adaptive mechanisms for TML running on MCUs.

9.4.1 Application Scenarios and Datasets

In this experimental section, the following two application scenarios have been considered:

- the *speech-command identification* scenario whose goal is to correctly recognize an user-speech command present in a one-second long audio clip. For this purpose, the *Synthetic Speech Commands Dataset* (Buchner, 2017; Warden, 2018) has been considered. This dataset comprises 30 classes of commands, corresponding, for example, to "*up*", "*left*", "*yes*", "*go*", or a number from "*zero*" to "*nine*". Moreover, the audio files within the dataset comprise different kinds of voices as well as different types of noisy classes.

- the *image classification* scenario whose goal is to classify an image containing exactly one object. The well-known ImageNet (Deng et al., 2009) dataset, comprising 1000 classes, has been considered.

9.4.2 The Considered Concept Drift affecting $\mathcal{P}$

Two different kinds of concept drift affecting the data-generation process $\mathcal{P}$ have been considered:

- the addition of noise on x. This type of concept drift models the scenario where a failure on the microphone acquiring the audio clip occurs. To achieve this goal, the noisy variant of each class within the *Synthetic Speech Commands Dataset* is considered after the change;

- a change in the classification problem, i.e., a variation in the set of classes Δ.

9.4.3 Experimental Settings

In this experimental analysis, the considered feature extractor ϱ refers to the first layer of the well-known ResNet-18 CNN (He et al., 2016). This layer comprises a convolutional layer with 64 7x7 three-dimensional filters with stride 2, a batch-normalization layer, a ReLU non-linearity, and a 3x3 max-pooling layer with stride 2. The dimensionality reduction operator ς discards 63 out of 64 filters by keeping only the one with the highest mean activation on the ImageNet benchmark. Consequently, the resulting DL model $\varsigma \circ \varrho$ is a single 7x7x3 filter that has 147 parameters and occupies 588B with a 32-bit floating-point representation.

In the *speech-command identification* scenario, the audio waveform (sampled at $f_a = 22050$ Hz) is converted into a spectrogram through a Short-Time Fourier Transform with

Table 9.4: *The impact of condensing techniques (C) in both the application scenarios and in stationary conditions, when no update is done. A SVM and Neural Network (with one fully-connected layer) classifiers have been added as baselines. The results represent the mean ± standard deviation accuracy (α_δ) and memory (m_δ) over 20 experiments with $\delta = \{2, 3, 5\}$ classes. The memory is expressed in terms of number of training samples within $\mathcal{T}$, with $|\mathcal{T}| = 100 \cdot \delta$.*

(a) *The condensing impact in the Speech Command Identification scenario.*

Algorithm	α_2	m_2	α_3	m_3	α_5	m_5
SVM	**0.93±0.05**	138±17	**0.88±0.04**	244±16	**0.81±0.05**	447±18
NN–FC1	0.71±0.14	2	0.75±0.06	3	0.49±0.12	5
kNN	0.89±0.06	200	0.82±0.06	300	0.74±0.07	500
kNN + C	0.88±0.06	**64±20**	0.81±0.04	**128±22**	0.73±0.06	**255±33**

(b) *The condensing impact in the Image Classification scenario.*

Algorithm	α_2	m_2	α_3	m_3	α_5	m_5
SVM	**0.73±0.07**	188±9	**0.61±0.05**	292±6	**0.46±0.05**	496±4
NN–FC1	0.60±0.08	2	0.45±0.05	3	0.29±0.05	5
kNN	0.66±0.07	200	0.49±0.07	300	0.33±0.06	500
kNN + C	0.66±0.07	**119±18**	0.51±0.06	**214±19**	0.35±0.04	**414±16**

windows of size $n_{fft} = 512$ and a step $h_l = 512$ and then converted into a colored one by means of a colormap. In the *image classification*, instead, the images are resized to 224x224x3 before being passed as input to $\mathcal{D}$. The resulting one-second audio has a memory footprint of 88 200B, the image 602 112B, whereas the resulting colored spectrogram of size 257x44x3 requires 135 696B.

The change always occurs after half of the available data, i.e., 500 samples per class in the *image classification scenario* and 750 in the *speech command identification* one. Finally, 100 samples per class are provided to all the algorithm as initial training set $\mathcal{T}$, i.e., $|\mathcal{T}| = 100 \cdot \delta$. Experimental results are averaged over twenty runs.

9.4.4 Evaluating Effects of the Pre-Processing Through Condensing

This section aims at studying the impact of condensing the kNN $\mathcal{K}$ training set $\mathcal{T}$ with Algorithm 6.2. To achieve this goal, the proposed TML-CD $\mathcal{D}$ is configured with the block $\varsigma \circ \varrho$ presented in Section 9.4.3 and an initial training set $\mathcal{T}$ with size $|\mathcal{T}| = 100 \cdot \delta$. Then, the classification capabilities of $\mathcal{D}$ when condensing is employed are evaluated in stationary conditions, i.e., no adaptation is carried out during the operational life of $\mathcal{D}$. In addition to $\mathcal{D}$ without or with the initial condensing (referred to as kNN and kNN+C in the following, respectively), the comparison comprises two well-known classifiers, i.e., a Support Vector Machine (SVM) and a single fully-connected layer neural network classifier (NN–FC1). Both the classifiers are applied on the same features of the kNN $\mathcal{K}$, i.e., those extracted by $\varsigma \circ \varrho$ (on the initial training set $\mathcal{T}$). Moreover, the SVM is trained until convergence, whereas the NN–FC1 is trained for 3 epochs with stochastic gradient descent, no momentum, and a learning rate $\eta \in \left[1e^{-2}, 5e^{-3}, 1e^{-3}, \ldots, 1e^{-5}\right]$. In our experiments, only the best performing NN–FC1 classifier is shown. It is crucial to stress that an unfeasible training procedure in MCUs characterizes both the SVM and the NN,

so they cannot be considered for the on-device training phase.

Table 9.4 shows the result in the proposed application scenarios with a different number of classes. The SVM and the NN–FC1 classifiers present the highest and worst accuracy in all the considered application scenarios, respectively. The proposed TML solution $\mathcal{D}$ (without condensing) shows accuracies smaller than the SVM classifier by 4 to 8% in *speech command identification* and 7 to 13% in *image classification* scenarios. As expected, condensing the training set $\mathcal{T}$ has a limited impact on the accuracy (at most 1% drop in *speech command identification* scenario). However, it allows to reduce the memory requirements significantly. Without considering the NN classifier, $\mathcal{D}$ with condensing is the algorithm providing the lowest memory demand.[17] In the *speech command identification* scenario, the number of stored samples indeed ranges from 32 to 50% of the provided samples (with δ from 2 to 5), representing the 46 to 57% of the ones required by the SVM, i.e., its support vectors. In the *image classification* scenario, the memory saving is significantly lower, with the SVM retaining almost all the samples as support vectors and the condensed $\mathcal{D}$ storing 60 to 82% of them. From now on, the proposed TML-CD $\mathcal{D}$ is assumed to always rely on condensing algorithm in the configuration and testing stages (when available).

9.4.5 Experimental Results in Presence of Concept Drift

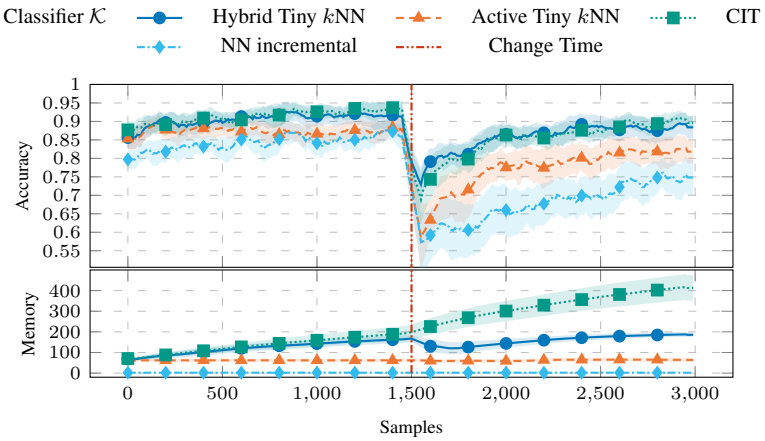
Figure 9.4 compares the three proposed adaptive algorithms, i.e., CIT, Active Tiny kNN, and Hybrid Tiny kNN, in the two considered scenarios with different numbers of classes δ. We considered two different figures of merit: the mean $\pm$ std *accuracy* (the curve of each experiment is the convolution of the correct predictions of each experiment with a 100-dimensional filter with all values of 0.01) and *memory footprint*, measured as the number of samples within the training set $\mathcal{T}$, over all the experiments. It is crucial to point out that the memory footprint does not include any other auxiliary source of memory, e.g., the history window W of the Active Tiny-kNN algorithm (that has a size of $\varpi = 100 \cdot \delta$).

As a comparison, this experimental analysis includes also a continuously learning single-layer fully-connected classifier (NN–FC1) operating on the features extracted by $\varsigma \circ \varrho$ and performing a back-propagation step for each incoming sample. The NN–FC1 is trained for 3 epochs with stochastic gradient descent, no momentum, and a learning rate η, with $\eta \in \left[1e^{-2}, 5e^{-3}, 1e^{-3}, \ldots, 1e^{-5}\right]$. Moreover, during the testing stage, the learning rate for back-propagation might be reduced ($\eta \in \left[1e^{-3}, 5e^{-4}, 1e^{-4}, \ldots, 1e^{-6}\right]$). Among all the possible combinations, Figure 9.4 shows only the one with the largest accuracy.

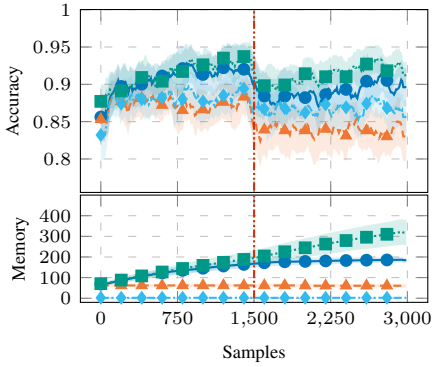
In more detail, Figure 9.4 shows the accuracy and memory footprint in five different configurations: the *Speech Command Identification scenario* where one-class changes with $\delta = \{2, 3, 5\}$ classes (Figures 9.4a, 9.4d, and 9.4e) and with the introduction of noise with $\delta = 2$ (Figure 9.4b), and the *Image Classification* scenario where one-class changes with $\delta = 2$ (Figure 9.4c).

The NN-FC1 baseline is the worst algorithm in almost all the cases, with low capabilities of recovering after change when $\delta > 2$ (Figures 9.4d and 9.4e). The noise has a limited impact on the accuracy, as shown in Figure 9.4b, whose small degradation is detected neither by the Active nor by the Hybrid Tiny kNN algorithms. With the other changes, the

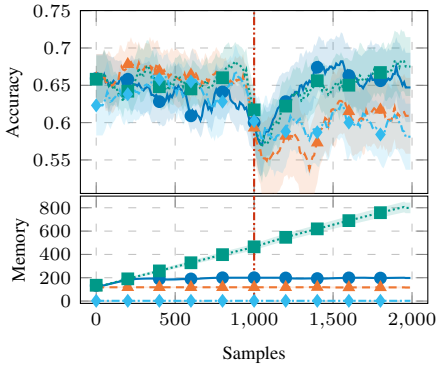
[17] The NN–FC1 memory footprint corresponds to that of its weights, which are equal to the number of classes multiplied by the size of classifier inputs. Since the input size is the same for all the classifiers, the NN-FC1 memory is that of δ samples.



(a) *2-class Speech Command Identification where a class changes after half samples.*



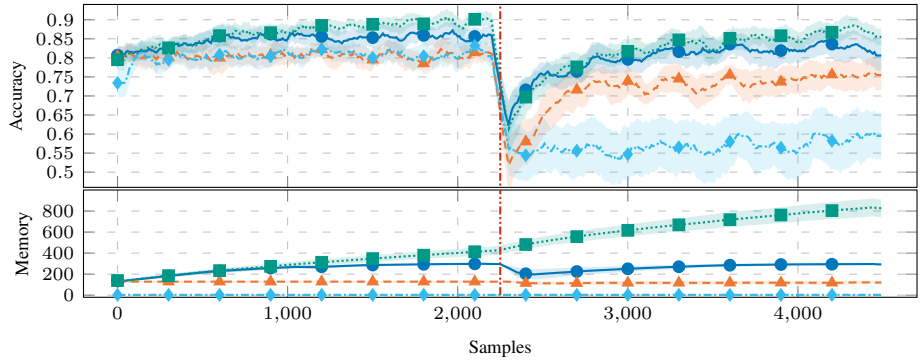
(b) *2-class Speech Command Identification where noise affects data after half samples.*



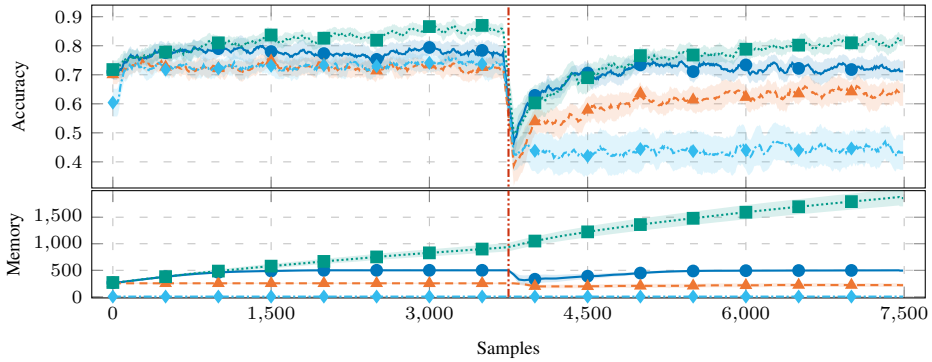
(c) *2-class Image Classification where a class changes after half samples.*

Figure 9.4: *The mean accuracy and the number of samples to be kept in $\mathcal{K}$'s training set $\mathcal{T}$ over time of the three proposed adaptive algorithms and a continuously learning Neural Network (with one fully-connected classifier). The plots represent the mean $\pm$ standard deviation over 20 experiments with $\delta = \{2, 3, 5\}$ classes and the considered scenarios/changes. The algorithms receive as input a training set $\mathcal{T}$, with $|\mathcal{T}| = 100 \cdot \delta$.*

proposed algorithms work as expected. On the one hand, the (passive) CIT algorithm continuously improves over time, at the expense of an unbounded memory growth (in these experiments, none of the approaches detailed in Section 6.3.1 to control it has been considered). Moreover, in all the considered scenarios, the slope of the samples' curve increases at the change time, highlighting the accuracy drop due to the change itself. On the other hand, the Active kNN algorithm is able to recover after a change keeping its memory footprint nearly constant and significantly lower than the size of history window W (not shown in the Figure 9.4) due to condensing. Finally, the Hybrid Tiny kNN algorithm combines the advantages of both the CIT and the Active Tiny kNN. It can recover faster than the two



(d) *3-class Speech Command Identification where a class changes after half samples.*



(e) *5-class Speech Command Identification where a class changes after half samples.*

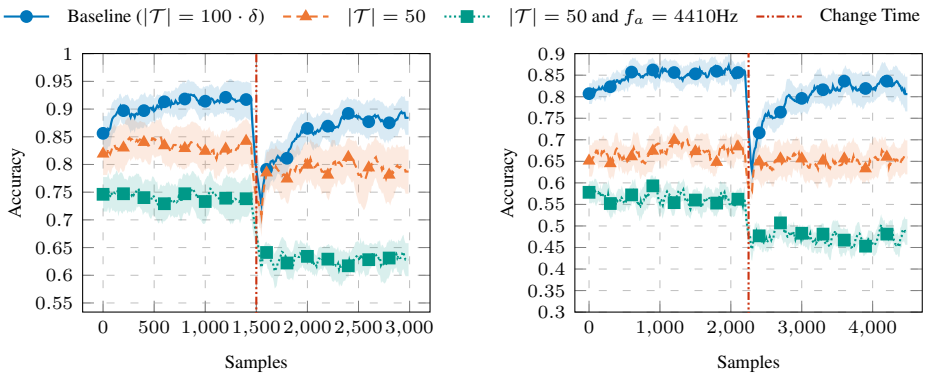
Figure 9.4 (Cont.): *The mean accuracy and the number of samples to be kept in $\mathcal{K}$'s training set $\mathcal{T}$ over time of the three proposed adaptive algorithms and a continuously learning Neural Network (with one fully-connected classifier). The plots represent the mean $\pm$ standard deviation over 20 experiments with $\delta = \{2, 3, 5\}$ classes and the considered scenarios/changes. The algorithms receive as input a training set $\mathcal{T}$, with $|\mathcal{T}| = 100 \cdot \delta$.*

other algorithms in all the considered scenarios and keep the memory footprint under control (by taking into account also the Active Tiny kNN history window memory footprint, the Hybrid Tiny kNN has the lowest footprint). Moreover, it shows the best accuracy, being overcome by CIT only when it saturates the maximum size of its training set $\mathcal{T}$ that is here fixed to $\varpi = 100 \cdot \delta$. This effect is visible in particular in Figures 9.4d and 9.4e.

9.4.6 Porting the Hybrid Tiny kNN on the STM32 MCUs

This section aims to show the technological feasibility of the Hybrid Tiny kNN algorithm in the *Speech command identification* scenario. To achieve this goal, we considered the following three different MCUs:

- The *STM32H743* is a high-performance MCU having a 480 MHz Cortex-M7 pro-



(a) *The results with $\delta = 2$ classes.*

(b) *The results with $\delta = 3$ classes.*

Figure 9.5: *The mean accuracy of the Hybrid Tiny kNN when satisfying the technological requirements of the STM32 boards. The plots represent the mean $\pm$ standard deviation over 20 experiments in the 2-class Speech Command Identification where a class changes after half samples. The algorithms receive as input a training set $\mathcal{T}$, with $|\mathcal{T}| = 50$.*

cessor, 1024 KB of RAM (split into five blocks of different speed), and 2048 KB of Flash memory;

- The *STM32F767* is a high-performance MCU having a 216 MHz Cortex-M7 processor, 512 KB of RAM, and 2048 KB of Flash memory;
- The *STM32F401* is a general-purpose MCU having a 84 MHz Cortex-M4 processor, 96 KB of RAM, and 512 KB of Flash memory.

The main technological constraint imposed by such board is the one on the memory, i.e., the maximum memory footprint of the Hybrid kNN algorithm cannot overcome the available RAM of each MCU (in the case of *STM32H743* that limit is lowered to 512KB, i.e., the size of the fastest RAM block). To satisfy this memory constraint, we set the maximum training set size of the Hybrid Tiny kNN algorithm to $|\mathcal{T}| = 50$. In addition, for the *STM32F401* board, the sampling frequency f_a is reduced from $f_a = 22050$Hz to $f_a = 4410$Hz as suggested in (Disabato et al., 2021a). This guarantees a strong reduction in the memory footprint of the input audio (17 640B), the generated spectrogram with windows of size $n_{fft} = 128$ and step $h_l = 128$ (65x35x3 occupying 27 300B), and the output of the feature extractor (17x9 occupying 588B). Table 9.5 details the memory footprint of the Hybrid Tiny kNN deployed for the STM32H743 and STM32F767 (Table 9.5a) and the STM32F401 (Table 9.5b).

Figure 9.5 shows the effects of such technological choices in the considered scenario when a class change after half samples with $\delta = \{2, 3\}$. In that figure, the baseline is the Hybrid Tiny kNN algorithm introduced in Section 6.3.3, where $\varpi = 100 \cdot \delta$. The algorithms deployed on the MCUs exhibit a constant accuracy until the change with a minimum gain due to passive adaptation (limited by the constraint on the training set size). The gap w.r.t. the baseline algorithm is between 4 and 10% (15% to 25% when varying

Table 9.5: *The detailed memory footprint (with a 32-bit data type) of the Hybrid Tiny kNN on the STM32 MCUs. The size of the training set $\mathcal{T}$ includes those of samples (see $\varsigma \circ \varrho$ output), their labels (32bit), and their timestamps (32bit).*

(a) *STM32H743 and STM32F767.*

	Size	Memory Footprint (B)
Audio ($t_a = 1$ s, $f_a = 22050$ Hz)	1x22050	88 200
Spectrogram ($n_{fft} = h_l = 512$)	257x44x3	135 696
$\varsigma \circ \varrho$ (1 convolutional filter 7x7x3)	7x7x3	588
$\varsigma \circ \varrho$ output	65x11	2 860
$\mathcal{K}$'s Training Set $\mathcal{T}$	50	143 400
Total		370 744

(b) *STM32F401.*

	Size	Memory Footprint (B)
Audio ($t_a = 1$ s, $f_a = 4410$ Hz)	1x4410	17 640
Spectrogram ($n_{fft} = h_l = 128$)	64x35x3	27 300
$\varsigma \circ \varrho$ (1 convolutional filter 7x7x3)	7x7x3	588
$\varsigma \circ \varrho$ output	17x9	612
$\mathcal{K}$'s Training Set $\mathcal{T}$	50	31 000
Total		77 140

Table 9.6: *The experimental execution times, measured in milliseconds, on three MCUs. The measured times are: the spectrogram processing t_p, the feature extraction $t_{\varsigma \circ \varrho}$, and the $\mathcal{K}$ prediction with 10 and 50 samples within $\mathcal{T}$. The time $t_{\mathcal{K},50}$ also shows the worst prediction time when the adaptation is involved.*

MCU	t_p	$t_{\varsigma \circ \varrho}$	$t_{\mathcal{K},10}$	$t_{\mathcal{K},50}$
STM32H743ZI	22.9	18.6	2.0	2.9–6.2
STM32F767ZI	53.8	41.5	2.2	4.0–12.1
STM32F401RE	34.6	43.1	2.2	4.6–136.0

also the acquisition frequency). After the change, the algorithm with $|\mathcal{T}| = 50$ recovers as fast as the baseline, then the effects of the passive updates create a gap in the accuracy. The algorithm that considers a variation also in the sampling frequency, instead, shows a larger drop in accuracy (about 10 to 15%). However, its passive updates are able to recover after the change, slightly improving the accuracy over time.

Table 9.6 reports the experimental execution times of the $\mathcal{D}$ blocks in the considered MCUs. More in detail, the measured quantities are: the processing time t_p needed to transform the acquired 1s audio into a spectrogram; the feature extractor and dimensionality reduction blocks' execution $t_{\varsigma \circ \varrho}$; and the kNN $\mathcal{K}$ prediction time $t_{\mathcal{K},|\mathcal{T}|}$ with two different cardinalities of its training set, 10 and 50 (the latter also shows the worst measured prediction time when an adaptation has been made). Results are particularly interesting. In particular, the processing and the feature extraction are the two predominant times, requiring 41.5ms and 95.3ms on high-performance MCUs (the STM32H7 and the STM32F7) and 77.7ms on a general-purpose one (the STM32F4, although on a smaller spectrogram).

The $\mathcal{K}$'s prediction and the adaptation, when employed, are negligible w.r.t. the other times, being the 7 and the 15% on the STM32H7, the 4 and the 13% on the STM32F7, and the 6 and the 175% on the STM32F4 (whose adaptation is the only exception). As a final remark, the total time required from the processing of the acquired audio to the final prediction, including the possible adaptations, is significantly lower than that of the acquisition, showing the effectiveness of the proposed Hybrid Tiny kNN algorithm on three real MCUs.

CHAPTER 10

Evaluating Deep and Wide Tiny Machine Learning Solutions

This chapter aims to collect the results about proposed solutions for Deep and Wide Tiny Machine Learning in Part III.

10.1 Evaluating the Methodology to Distribute Deep Learning Models over an IoT System

The methodology –presented in the paper (Disabato et al., 2021b) and described in Chapter 7– to distribute the computation of Deep Learning Models over an IoT System has been validated on four DLMs and four families of IoT devices in a synthetic scenario of distributed image classification for the control of a critical area (e.g., recognition of the presence of target objects in a given area through image classification). The monitored area is assumed to be a 30m square, and the positions of the IoT units, as well as those of the sources s_us and the sink f, are randomly selected following a uniform distribution. The hyper-parameter L, setting the maximum number of CNN layers per IoT unit, ranges from 1 to 5.

This section is organized as follows. Section 10.1.1 recaps the employed off-the-shelf IoT units and their transmission technologies, whereas Section 10.1.2 describes the figure of merit. Then, Sections 10.1.3 and 10.1.4 describe the experimental results in two different synthetic IoT systems, whereas Section 10.1.5 presents the results in a real IoT

system.

10.1.1 The considered IoT Units

Table 7.1c details the four considered IoT units: the STM32H7, a simple microcontroller unit endowed with a 400 MHz-Cortex M7, 1 MB of RAM, and no operating system; the Raspberry Pi 3B+ equipped with 1GB of RAM and a 1.4GHz quad-core ARM Cortex A53; the OrangePi Zero with 256MB of RAM and a 1.2GHz quad-core ARM Cortex A7; and the BeagleBone AI with 1GB of RAM and a 1.5GHz dual-core ARM Cortex A15.

The maximum memory usage $\bar{m}_i$s has been defined as half of the available RAM, i.e., 512KB for the STM32H7, 128MB for the OrangePi Zero, and 512MB for both the BeagleBone AI and the Raspberry Pi3B+. The number of multiplications per second e_is has been defined as a tenth of the clock cycles (per number of cores). Hence, $e = 40$ for the STM32H7, $e = 300$ for the BeagleBone AI (150 per core), $e = 480$ for the OrangePi Zero (120 per core, if we consider the maximum frequency of 1.2GHz), and $e = 560$ for the Raspberry 3B+ (140 per core). The constraints on the computational load $\bar{c}_i$s have not been considered since they are application-specific.

The transmission technologies the IoT units are equipped with are the *Wi-Fi 4* (standard IEEE 802.11n) and *Wi-Fi HaLow* (standard IEEE 802.11ah). The transmission range is $d_t = 7.5m$ (a tenth of the minimum indoor range). The Wi-Fi 4 data-rate is $\rho = 72.2$ Mb/s, that corresponds to the single-antenna scenario with 64-QAM modulation on the 20 MHz channel (Xiao, 2005), whereas the *Wi-Fi HaLow* one is $\rho = 7.2$ Mb/s with a single-antenna and 64-QAM modulation on the 2 MHz channel (Adame et al., 2014).

10.1.2 Figures of Merit

The methodology presented in Chapter 7 is evaluated on the "data production to decision making"-latency t defined as the time between input acquisition (at a source s) and classification reception at f. To further clarify the effects of data communication and computation, t is split into the transmission t_t and the processing t_p terms. The former term refers to the sum of all transmission times (from a source to IoT units, between IoT units, or from IoT units to the target unit f); the latter sums the processing times on the IoT units. These terms are computed as defined in Section 7.2, whereas additional sources of delays, such as transmission handshakes or repeated transmissions (due to failures), have been neglected.

For each setting, transmission technology, and configuration, the evaluated figure of metric is the mean $\pm$ standard deviation of each latency term, i.e., t, t_t, and t_p, computed on 500 randomly generated IoT systems.

It is noteworthy to point out that the accuracy has not been considered as a metric since the proposed method does not introduce any approximation w.r.t the original CNN. Hence there is no accuracy loss due to the placement of the CNN layers.

10.1.3 First IoT System: 30 IoT Units and Two Families

The first IoT system comprises $N = 30$ IoT units belonging to two technological families, i.e., the STM32H7 and the Raspberry Pi 3B+, with two settings for the IoT units partitioning, i.e., 50%–50% and 90%–10%.

Table 10.1: *The methodology results with $N = 30$ STM32H7 and Raspberry Pi 3B+ IoT units in the 50%-50% configuration scenario and a single (EX–)DLM te be placed. The figure of merit (mean $\pm$ std) is the latency t (transmission t_t plus processing t_p).*

(a) *The results with the Wi-Fi 4 as adopted transmission technology.*

		5/6-layer DLM's Latency t (ms)			(EX–)AlexNet's Latency t (ms)		
	L	t_t	t_p	$t = t_t + t_p$	t_t	t_p	$t = t_t + t_p$
No Early–Exit	1	7.49 ± 0.36	44.93 ± 0.00	52.42 ± 0.36	203.06 ± 36.85	1257.71 ± 0.00	1460.77 ± 36.85
	2	1.78 ± 0.21	44.93 ± 0.00	46.71 ± 0.21	127.81 ± 28.49	1257.71 ± 0.00	1385.52 ± 28.49
	3	0.48 ± 0.14	44.93 ± 0.00	45.41 ± 0.14	98.75 ± 26.63	1257.71 ± 0.00	1356.46 ± 26.63
	4	0.37 ± 0.11	44.93 ± 0.00	45.30 ± 0.11	95.82 ± 26.54	1257.71 ± 0.00	1353.53 ± 26.54
	M	0.28 ± 0.11	44.93 ± 0.00	45.21 ± 0.11	72.49 ± 24.08	1257.71 ± 0.00	1330.20 ± 24.08
Early–Exit	1	5.87 ± 0.27	7.27 ± 0.00	13.14 ± 0.27	129.42 ± 28.29	598.92 ± 81.81	728.35 ± 86.17
	2	0.34 ± 0.10	7.27 ± 0.00	7.61 ± 0.10	93.70 ± 24.49	596.19 ± 0.00	689.89 ± 24.49
	3	0.29 ± 0.10	7.27 ± 0.00	7.57 ± 0.10	72.10 ± 22.08	596.19 ± 0.00	668.29 ± 22.08
	4	0.28 ± 0.10	7.27 ± 0.00	7.55 ± 0.10	72.07 ± 22.08	596.19 ± 0.00	668.26 ± 22.08
	M	0.28 ± 0.10	7.27 ± 0.00	7.55 ± 0.10	71.84 ± 22.07	596.19 ± 0.00	668.03 ± 22.07

(b) *The results with the Wi-Fi HaLow as adopted transmission technology.*

		5/6-layer DLM's Latency t (ms)			(EX–)AlexNet's Latency t (ms)		
	L	t_t	t_p	$t = t_t + t_p$	t_t	t_p	$t = t_t + t_p$
No Early–Exit	1	74.82 ± 3.94	45.16 ± 0.57	119.98 ± 4.06	2042.48 ± 401.64	1265.89 ± 141.47	3308.37 ± 432.91
	2	17.75 ± 2.08	44.93 ± 0.00	62.68 ± 2.08	1298.23 ± 332.66	1263.16 ± 115.58	2561.39 ± 361.81
	3	4.49 ± 0.89	45.08 ± 0.46	49.57 ± 1.03	1009.79 ± 334.87	1260.44 ± 81.77	2270.22 ± 351.10
	4	3.64 ± 0.87	44.93 ± 0.00	48.57 ± 0.87	976.71 ± 317.48	1263.16 ± 115.58	2239.87 ± 348.56
	M	2.80 ± 0.88	44.93 ± 0.00	47.73 ± 0.88	750.02 ± 322.44	1260.44 ± 81.77	2010.46 ± 339.52
Early–Exit	1	58.72 ± 2.47	7.27 ± 0.01	65.99 ± 2.47	1304.74 ± 310.87	598.93 ± 81.91	1903.66 ± 324.56
	2	3.53 ± 1.15	7.27 ± 0.00	10.80 ± 1.15	947.04 ± 258.12	596.19 ± 0.00	1543.23 ± 258.12
	3	3.04 ± 1.10	7.27 ± 0.00	10.31 ± 1.10	728.19 ± 221.12	596.19 ± 0.00	1324.38 ± 221.12
	4	2.90 ± 1.10	7.27 ± 0.00	10.17 ± 1.10	727.87 ± 221.10	596.19 ± 0.00	1324.06 ± 221.10
	M	2.88 ± 1.10	7.27 ± 0.00	10.16 ± 1.10	725.53 ± 220.95	596.19 ± 0.00	1321.72 ± 220.95

Single Deep Learning Model Configuration. This configuration encompasses a single DLM, either with or without an Early-Exit, to be placed on the considered IoT system. The methodology is tested with $L \in \{1, 2, 3, 4\}$ and compared to the Cloud approximation ($L = M$) where all the computation can be placed on a single node, i.e., it can be seen as an approximation of sending data directly to the Cloud and then receiving back the result.

Table 10.1 shows the results in the partition setting 50%–50%. Interesting results arise. First of all, the expected processing time t_p is significantly reduced when the Early Exit is employed, as expected and studied in (Bolukbasi et al., 2017; Disabato and Roveri, 2018) for all the considered DLMs.[18] In the case of 6-layer EX-DLM, the Early-Exit allows to save about 75-84% (45 to 80% with *Wi-Fi HaLoW*) of the latency t, whereas on the AlexNet 34 to 50%.

After that, with *Wi-Fi 4* the transmission latency t_t is significantly lower than the processing one t_p. Thus it is reasonable to assume that the achieved t_p is the minimum feasible in this IoT system. However, with *Wi-Fi HaLow*, the minimum experimental la-

[18]The latency t and its terms t_t and t_p, are defined as an expected value with EX-DLMs, by weighting their values up to each layer j of DLM u with the probability $g_{u,j}$ of providing the classification.

Table 10.2: *The methodology results with $N = 30$ STM32H7 and Raspberry Pi 3B+ IoT units and two 5-layer DLMs (Figure 7.1a). The figure of merit (mean $\pm$ std) is the latency t (transmission t_t plus processing t_p) and is summed over all the DLMs.*

(a) *The results with the Wi-Fi 4 as adopted transmission technology.*

		50 – 50 Latency t (ms)			90 – 10 Latency t (ms)		
	L	t_t	t_p	$t = t_t + t_p$	t_t	t_p	$t = t_t + t_p$
No shared layers	1	15.6 ± 1.1	89.9 ± 0.1	105.4 ± 1.1	19.4 ± 5.5	634.4 ± 417.0	653.8 ± 413.6
	2	3.8 ± 0.6	89.9 ± 0.0	93.6 ± 0.6	10.2 ± 5.0	317.1 ± 401.5	327.3 ± 399.7
	3	1.1 ± 0.4	89.9 ± 0.0	90.9 ± 0.4	4.0 ± 2.9	198.0 ± 240.0	201.9 ± 242.4
	4	0.8 ± 0.3	89.9 ± 0.0	90.7 ± 0.3	4.0 ± 4.5	119.8 ± 67.0	123.7 ± 71.2
	5	0.7 ± 0.3	89.9 ± 0.0	90.5 ± 0.3	3.0 ± 2.8	105.1 ± 34.1	108.1 ± 36.6
First two layers shared	1	15.4 ± 1.2	89.9 ± 0.1	105.3 ± 1.2	23.4 ± 9.0	452.8 ± 470.8	476.2 ± 467.4
	2	3.7 ± 0.5	89.9 ± 0.0	93.5 ± 0.5	9.6 ± 8.6	250.6 ± 389.8	260.2 ± 388.0
	3	2.2 ± 0.4	89.9 ± 0.0	92.1 ± 0.4	7.1 ± 5.2	115.9 ± 63.1	122.9 ± 65.8
	4	0.9 ± 0.3	89.9 ± 0.0	90.8 ± 0.3	2.3 ± 1.6	90.5 ± 1.3	92.8 ± 2.2
	5	0.8 ± 0.2	89.9 ± 0.0	90.7 ± 0.2	2.0 ± 1.4	90.2 ± 0.6	92.2 ± 1.7

(b) *The results with the Wi-Fi HaLow as adopted transmission technology.*

		50 – 50 Latency t (ms)			90 – 10 Latency t (ms)		
	L	t_t	t_p	$t = t_t + t_p$	t_t	t_p	$t = t_t + t_p$
No shared layers	1	154.4 ± 11.5	91.2 ± 6.1	245.6 ± 14.1	187.0 ± 41.4	624.6 ± 416.9	811.6 ± 394.1
	2	37.4 ± 5.3	90.0 ± 0.6	127.4 ± 5.4	95.3 ± 44.4	301.7 ± 383.1	397.0 ± 372.3
	3	9.9 ± 2.7	90.4 ± 0.9	100.3 ± 3.0	32.9 ± 28.8	188.1 ± 228.8	221.0 ± 254.2
	4	8.1 ± 2.6	89.9 ± 0.0	98.0 ± 2.6	37.5 ± 42.6	116.6 ± 64.0	154.1 ± 105.1
	5	6.4 ± 2.7	89.9 ± 0.0	96.3 ± 2.7	28.6 ± 27.0	103.5 ± 32.6	132.1 ± 57.2
First two layers shared	1	152.3 ± 9.8	90.7 ± 1.2	243.1 ± 10.1	223.7 ± 63.6	454.4 ± 459.1	678.0 ± 434.8
	2	36.0 ± 4.3	90.2 ± 0.9	126.2 ± 4.5	85.4 ± 72.2	252.5 ± 389.8	338.0 ± 381.7
	3	21.9 ± 3.9	90.1 ± 0.6	112.1 ± 4.0	65.3 ± 46.3	117.0 ± 62.8	182.3 ± 97.5
	4	9.0 ± 2.2	90.2 ± 0.9	99.2 ± 2.4	17.9 ± 14.1	92.1 ± 1.5	110.0 ± 14.4
	5	8.2 ± 2.2	90.0 ± 0.5	98.2 ± 2.2	17.1 ± 14.1	91.0 ± 0.7	108.0 ± 14.2

tency ($t_p = 44.93$ms for the 5-layer DLM and $t_p = 1257.71$ms for the AlexNet) cannot be achieved and, in particular, the AlexNet processing with $L = 1$ requires more than 3 seconds (instead of 1.46s with the Wi-Fi 4 transmission technology), a latency that might be unfeasible in many real applications.

The third crucial comment is about the $L = M$ case. The latency t of this case and those of corresponding ones having $L \geq 2$ ($L > 2$ with *Wi-Fi HaLow*) are almost equal (with $L = 1$ and *Wi-Fi 4* there is a 10% increment), showing the capability of the proposed methodology to distribute the DLM computation among the IoT units with negligible latency increments.

Multi Deep Learning Models Configuration. In this configuration, two 5-layer DLMs have to be placed on the considered IoT system, with and without the first two layers shared. Table 10.2 presents the results for all the configurations and transmission technologies. Several comments arise. At first, in the configuration 90%–10%, the methodology has to often rely on STM32H7 nodes. Hence the processing time is significantly increased (the computation capability e of a Raspberry is 14 times greater than that of an STM32H7). This result is even more evident when there are shared layers since the methodology can place less computation on STM32H7s.

The Wi-Fi 4 guarantees transmission latencies t_ts negligible w.r.t. the processing time t_p, that represents more than 85% of latency t (up to 96-99% with $L \geq 2$). Interestingly, the processing time is always equal to 89.9ms, which is the experimental minimum achievable value in this IoT system. This consideration is no longer valid with the *Wi-Fi HaLow*, where the two terms are comparable, especially when $L = 1$. The methodology cannot indeed always achieve the minimum experimental processing latency, but sometimes it has to rely on nearby STM32H7 units, as highlighted by the non-null standard deviation of the t_p, in the configurations with at least 50% of Raspberry Pi 3B+ units. Interestingly, despite the fact that the data-rate of the *Wi-Fi 4* is ten times greater than that of *Wi-Fi HaLow*, the latency t in the harsher case with 90% of STM32H7 units is similar for both the transmission technologies, with a maximum increment of 20% with $L = 1$ and no shared layers.

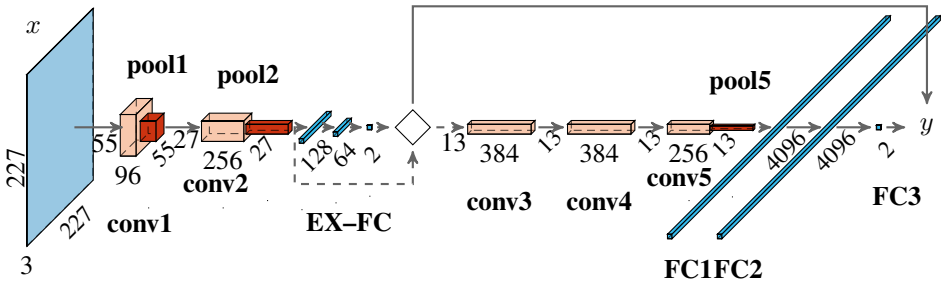
Finally, in all cases with $L \geq 2$ ($L > 2$ with *Wi-Fi HaLow*) the total latency t is comparable to the case ($L = M$), as in single DLM configuration. It is crucial to point out the importance of this result because distributing the DLM processing on various IoT units with a negligible increment in latency t will allow defining a pipeline in processing sequence of inputs. Indeed, when a unit has carried out the processing of DLM layers is designed to and sent the computed representation to the subsequent node, it is ready to operate on the next input, as in processor pipelines. Hence, the throughput of DLM processing can be significantly increased by processing inputs in a pipeline, with the bottleneck in the IoT unit responsible for the highest processing time.

10.1.4 Second IoT System: 50 IoT Units and Three Families

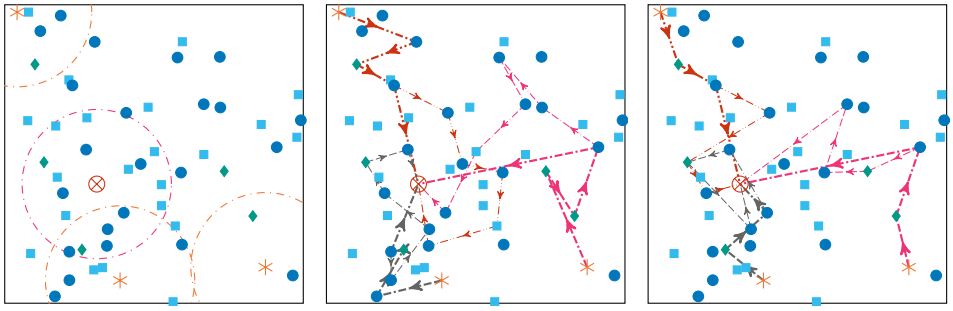
A second IoT system has $N = 50$ units belonging to three different technological families equipped with the *Wi-Fi 4* transmission technology and partitioned as follows: 45% of OrangePi Zero, 45% of BeagleBone AI, and 10% of Raspberry Pi 3B+. Figure 10.1b shows an example of this IoT system, where blue circles represent the OrangePi Zero units, cyan squares the BeagleBone AI units, green diamonds the Raspberry Pi 3B+ units, orange asterisks the sources, and a red circled cross the target unit.

The scenario is interesting because the most powerful IoT units in terms of both memory and computation capabilities, i.e., the Raspberry Pi 3B+, are just a few (about 5 in each simulated IoT system). In contrast, the remaining IoT units are characterized by contrasting peculiarities: on the one hand, the BeagleBone AI units have the same memory capacity as the Raspberry Pi 3B+ but only 54% of the computation one; on the other hand, the OrangePi Zero have almost the same computation capability of the Raspberry Pi 3B+ (85%, 160% if compared to BeagleBone AI), but only a fourth of the memory capability. It is worth noting that both the Raspberry Pi 3B+ and BeagleBone AI can store all the layers of the considered DLMs, whereas the OrangePi Zero units with 128 MB of RAM cannot store all the layers of (EX–)AlexNets. Consequently, a balancing between the faster OrangePi Zero units (in terms of computation capability) and the slower but with higher memory capacity BeagleBone AI units is expected in this IoT system (at least after all the Raspberry Pi 3B+ units have been considered, if enough closer).

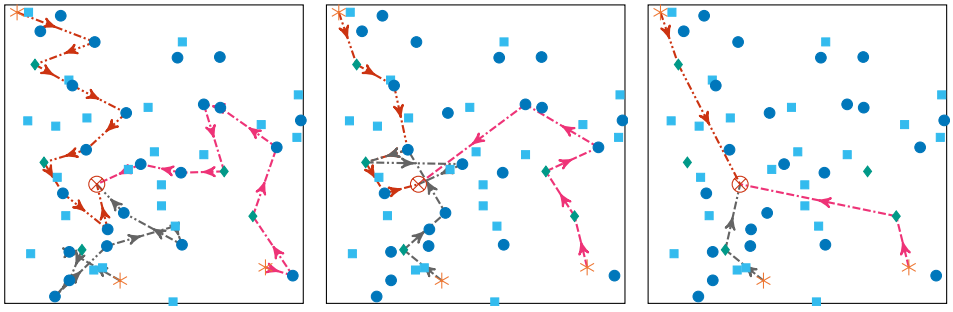
In addition to the figures of merit presented in Section 10.1.2, the number of considered nodes η_x is taken into account, where x is a technological family of IoT units, i.e., x can be R, O, or B, representing the Raspberry Pi 3B+, the OrangePi Zero, and the BeagleBone AI units, respectively.



(a) *The architecture of the Ex-AlexNet detailed in Table 7.1b, where x is the input image. The Early-Exit (composed of three fully-connected layers) can either provide the final output (solid line from the "decision" diamond) or further exploring the pipeline (dashed path, highlighting that the forwarded features are those in input to the Early-Exit layer).*



(b) *The considered IoT System.* **(c)** *The outcome with $L = 1$ and 3 EX-AlexNet.* **(d)** *The outcome with $L = 2$ and 3 EX-AlexNet.*



(e) *The outcome with $L = 1$ and 3 AlexNet.* **(f)** *The outcome with $L = 2$ and 3 AlexNet.* **(g)** *The outcome with $L = M$ and 3 AlexNet.*

Figure 10.1: *An example of the methodology outcome on a IoT system comprising $N = 50$ IoT units, i.e. Raspberry Pi 3B+, OrangePi Zero, and BeagleBone AI (10%, 45%, and 45% the probability of each kind of unit) and three (EX–)AlexNets to be placed. The sources are indicated by a star, whereas the target unit by a circled cross. The transmission range is indicated only for the sources and the target and is equal for all nodes. When EX–AlexNets (EX–A) are employed, the path with probability 0.772 is indicated by the thicker line, whereas the* full *path continues with the thin one.*

Table 10.3: *The methodology results on 500 randomly generated IoT systems with $N = 50$ OrangePi Zero, BeagleBone AI, and Raspberry Pi 3B+ units, with probability 45%-45%-10%, and various combinations of DLMs or EX-DLMs. The figures of merit (mean $\pm$ std) are the latency t (transmission t_t plus processing t_p) and the number η of IoT units used.*

(a) *The placement results of 1 AlexNet.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	206.22 ± 26.04	$1\,390.95 \pm 54.58$	$1\,597.17 \pm 54.33$	3.97 ± 1.51	4.01 ± 1.48	0.02 ± 0.15
2	151.25 ± 30.93	$1\,344.36 \pm 56.70$	$1\,495.61 \pm 60.89$	3.00 ± 0.87	1.43 ± 1.08	0.01 ± 0.12
3	125.55 ± 33.97	$1\,328.63 \pm 51.73$	$1\,454.19 \pm 62.16$	2.47 ± 0.63	0.64 ± 0.74	0.01 ± 0.12
4	119.76 ± 34.24	$1\,323.49 \pm 47.45$	$1\,443.25 \pm 59.89$	1.96 ± 0.50	0.51 ± 0.73	0.01 ± 0.09
M	97.88 ± 39.57	$1\,311.71 \pm 44.03$	$1\,409.58 \pm 66.63$	1.00 ± 0.11	0.17 ± 0.45	0.01 ± 0.09

(b) *The placement results of 3 AlexNets.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	578.58 ± 50.35	$4\,390.39 \pm 104.53$	$4\,968.97 \pm 119.87$	5.07 ± 2.01	17.89 ± 1.72	1.05 ± 1.24
2	410.26 ± 43.87	$4\,211.45 \pm 148.26$	$4\,621.71 \pm 142.12$	5.05 ± 1.97	7.72 ± 1.95	0.11 ± 0.38
3	350.06 ± 45.97	$4\,108.70 \pm 154.98$	$4\,458.76 \pm 152.25$	4.84 ± 1.71	4.65 ± 1.98	0.04 ± 0.26
4	349.40 ± 54.44	$4\,035.36 \pm 146.02$	$4\,384.76 \pm 147.68$	4.46 ± 1.34	3.13 ± 1.86	0.04 ± 0.26
M	306.24 ± 76.75	$3\,947.47 \pm 106.77$	$4\,253.71 \pm 144.39$	2.86 ± 0.46	0.90 ± 1.33	0.02 ± 0.21

(c) *The placement results of 4 AlexNets.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	772.23 ± 54.88	$5\,960.88 \pm 138.87$	$6\,733.11 \pm 156.94$	5.22 ± 2.20	21.73 ± 2.80	5.04 ± 2.99
2	521.34 ± 36.80	$5\,718.77 \pm 181.19$	$6\,240.11 \pm 175.43$	5.22 ± 2.20	11.62 ± 2.20	0.19 ± 0.55
3	437.76 ± 50.07	$5\,572.27 \pm 206.36$	$6\,010.03 \pm 196.11$	5.17 ± 2.12	7.47 ± 2.34	0.07 ± 0.40
4	441.12 ± 59.81	$5\,458.45 \pm 213.15$	$5\,899.57 \pm 201.50$	4.99 ± 1.85	5.25 ± 2.36	0.10 ± 0.38
M	401.83 ± 77.75	$5\,288.65 \pm 150.73$	$5\,690.48 \pm 183.33$	3.64 ± 0.74	1.81 ± 1.93	0.04 ± 0.30

The placement of 1 to 4 (EX–)AlexNet. Table 10.3 and in Figure 10.1 show the results of one to four (EX–)AlexNets placed in this IoT system. It is worth noting that with $L = M$, the layers of an (EX–)AlexNet can be placed on a single node if and only if Raspberry Pi 3B+ or BeagleBone AI IoT units are employed. If the methodology selects an OrangePi unit, at least another IoT unit must be considered to place all the layers, due to OrangePi memory constraints. The methodology tried indeed to avoid such option, as shown by the values of η_B almost equal to zero.

Several comments arise. First of all, the latency t of 1 AlexNet is higher than that in the IoT system comprising only Raspberry Pi 3B+ and STM32H7 (see Section 10.1.3). The reason resides in the fact that, in this IoT system, the probability that an IoT unit is a Raspberry is 10% –instead of 50%– and both the OrangePi Zero and the BeagleBone AI units have a smaller computation capability.

Second, the latency t of 3 AlexNets is close to that with 1 AlexNet multiplied by 3 (Tables 10.3a –10.3b). In fact, the difference in percentage ranges from 0.6% to 3.5%

Table 10.3 (Cont.): *The methodology results on 500 randomly generated IoT systems with $N = 50$ OrangePi Zero, BeagleBone AI, and Raspberry Pi 3B+ units, with probability 45%-45%-10%, and various combinations of DLMs or EX-DLMs. The figures of merit (mean ± std) are the latency t (transmission t_t plus processing t_p) and the number η of IoT units used.*

(d) *The placement results of 1 EX-AlexNet.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	117.54 ± 17.66	666.38 ± 33.56	783.92 ± 36.27	4.08 ± 1.46	3.79 ± 1.45	0.13 ± 0.34
2	93.87 ± 24.95	643.84 ± 34.46	737.71 ± 40.10	3.09 ± 0.82	1.51 ± 1.00	0.07 ± 0.25
3	77.03 ± 25.44	641.06 ± 34.80	718.09 ± 41.04	2.66 ± 0.58	0.44 ± 0.67	0.02 ± 0.13
4	74.56 ± 24.62	640.61 ± 33.94	715.17 ± 40.58	1.98 ± 0.49	0.44 ± 0.71	0.00 ± 0.06
M	68.15 ± 24.14	639.05 ± 34.41	707.20 ± 42.28	1.00 ± 0.06	0.24 ± 0.49	0.01 ± 0.08

(e) *The placement results of 3 EX-AlexNets.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	335.42 ± 21.31	$2\,078.03 \pm 54.60$	$2\,413.45 \pm 59.52$	4.95 ± 2.08	17.26 ± 1.72	1.79 ± 1.29
2	268.92 ± 38.28	1989.59 ± 72.05	2258.52 ± 72.76	4.91 ± 2.03	8.36 ± 1.82	0.55 ± 0.78
3	222.56 ± 39.05	1971.77 ± 75.12	2194.33 ± 75.51	4.79 ± 1.90	4.84 ± 2.22	0.17 ± 0.59
4	222.98 ± 39.87	1946.47 ± 73.10	2169.45 ± 76.50	4.37 ± 1.43	3.37 ± 2.00	0.11 ± 0.36
M	211.27 ± 42.69	1914.79 ± 64.03	2126.06 ± 75.80	2.87 ± 0.51	1.08 ± 1.36	0.03 ± 0.17

(f) *The placement results of 4 EX-AlexNets.*

	Latency t (ms)			Node usage η		
L	t_t	t_p	$t = t_t + t_p$	η_R	η_O	η_B
1	452.33 ± 29.81	2803.39 ± 67.85	3255.73 ± 78.62	4.95 ± 2.17	21.52 ± 2.72	5.53 ± 2.56
2	346.37 ± 43.73	2689.37 ± 94.61	3035.74 ± 93.17	4.95 ± 2.17	12.39 ± 1.79	1.19 ± 1.35
3	284.80 ± 46.16	2666.04 ± 100.07	2950.84 ± 96.52	4.93 ± 2.13	8.07 ± 2.15	0.72 ± 1.05
4	289.94 ± 44.66	2619.44 ± 103.84	2909.38 ± 102.62	4.79 ± 1.89	5.59 ± 2.24	0.25 ± 0.68
M	281.67 ± 50.74	2558.70 ± 98.07	2840.36 ± 105.54	3.63 ± 0.95	1.86 ± 2.12	0.12 ± 0.51

($L = M$ to $L = 1$), with the error percentage between the latency t_{3A} of placing 3 AlexNets and the latency t_A of placing 1 AlexNet multiplied by 3 computed as

$$\varepsilon_{t_{3A},3\cdot t_A} = \frac{t_{3A} - 3 \cdot t_A}{t_{3A}}. \tag{10.1}$$

When 4 AlexNets are employed (Table 10.3c), the range is slightly higher, i.e., 1 to 5% on the same values of L. This result shows the effectiveness of the proposed methodology in placing the DLMs in the given IoT system. Moreover, by observing the values of η_R, η_O, η_B it is clear that, whenever possible, the methodology relies on the fastest units, as expected by the fact that the transmission latency t_t is, in all the cases, significantly smaller than the processing t_p one.

Finally, as commented in Section 10.1.3, the latency t with $L = 1$ and $L = 2$ is close to that with $L = M$, with an increment ranging from 13% to 18%, and from 6% to

Table 10.4: *An experimental benchmark where a 5-layer DLM (Figure 7.1a) has to be placed on a IoT System having a Raspberry Pi 3B+ and two STM32H7s. The IoT units are equally spaced and equipped with Wi-Fi 4. With $L = 4$, the methodology outcome places the first four layers on the Raspberry and the last to one of the two STM32H7. The figure of merit is the latency t (transmission t_t plus processing t_p), expressed in milliseconds.*

L	Case	Wi-Fi 4 Latency (ms)		
		t_t	t_p	$t = t_t + t_p$
4	Model	0.37	44.98	45.35
	Experimental	0.42	68.47	68.89

9%, with 1 and 4 AlexNets, respectively, allowing us to define processing pipelines in the considered IoT system, to further reduce the latency t.

Tables 10.3d–10.3f investigates the same IoT scenario with 1, 3, and 4 EX-AlexNets to be placed. The latency t and its components t_p and t_t are defined as an expected value, by weighting the latency of each possible path within the EX-DLM by its probability. The mean numbers of nodes used η_R, η_O, and η_B are instead computed on the longest path within the DLM.

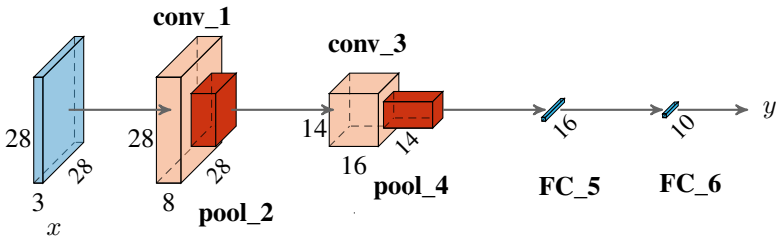
The trend in the results is analogous to the case with AlexNets, with smaller errors. The difference in percentage between placing 3 and 4 EX-AlexNets and 1 EX-AlexNet multiplied by 3 and 4 ranges from 0.2% to 2.5% and from 0.4% to 3.7% ($L = M$ to $L = 1$), respectively. Interestingly, the values of η_B are higher in this group of experiments, showing that the methodology more often relies on closer BeagleBone AI units to place part of the EX-AlexNet computation, reasonably on the less probable path.

10.1.5 Distribute a 5-layer DLM in a Real IoT System

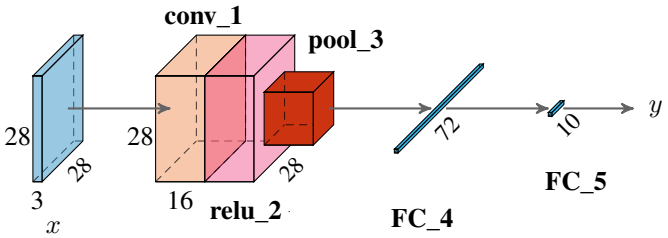
The methodology has been also applied to the placement of the 5-layer DLM (Figure 7.1a) on a real technological scenario comprising two *STM32H7*s and one *Raspberry Pi 3B+*. The transmission technology is the *Wi-Fi 4* and the connectivity is provided locally by the GL.iNet GL-MT300N-V2 router. Moreover, the hyper-parameter L has been set to 4, and the IoT units are equally spaced, i.e., at distance 1 from each other.

This experiment aims to compare the figures of merit t, t_t, and t_p estimated by the methodology with the corresponding measured values on the field. The outcome of the methodology assigned the first four layers of the CNN to the Raspberry and the fifth layer to one of the two STM32H7s. Table 10.4 shows the measured transmission and processing times. Interestingly, the experimental transmission time t_t is almost equal to the methodology estimation, whereas the experimental processing time t_p is 30% larger. This is justified by the fact that the model considered only multiplications. More in detail, the first four layers on Raspberry Pi 3B+ took 68.29 ms instead of 44.93 ms, whereas the fifth layer on STM32H7 176 μs (32 μs with code optimization) instead of 50 μs.

Eventually, the measured processing time t_p of the AlexNet on the Raspberry Pi 3B+ (median over 100 runs) is 1119.47 ms. In contrast, the one provided by the methodology is 1257.71 ms, showing that the model presented in Chapter 7 well describes this technological scenario.



(a) *The architecture of the 6-layer CNN.*



(b) *The architecture of the 5-layer CNN.*

Figure 10.2: *The CNNs provided by the HE-DL architecture in the experimental evaluation, where x represents the input image.*

10.2 Evaluating the HE-DL Architecture

The aim of this section is to evaluate the accuracy and the computation load of the deep-learning-as-service presented in the paper (Disabato et al., 2020) and described in Chapter 8. Section 10.2.1 details the CNNs provided by the deep-learning-as-service, whereas Section 10.2.2 describes the considered datasets. Finally, Sections 10.2.3–10.2.5 present the accuracy and computational load on both recall and transfer learning modality.

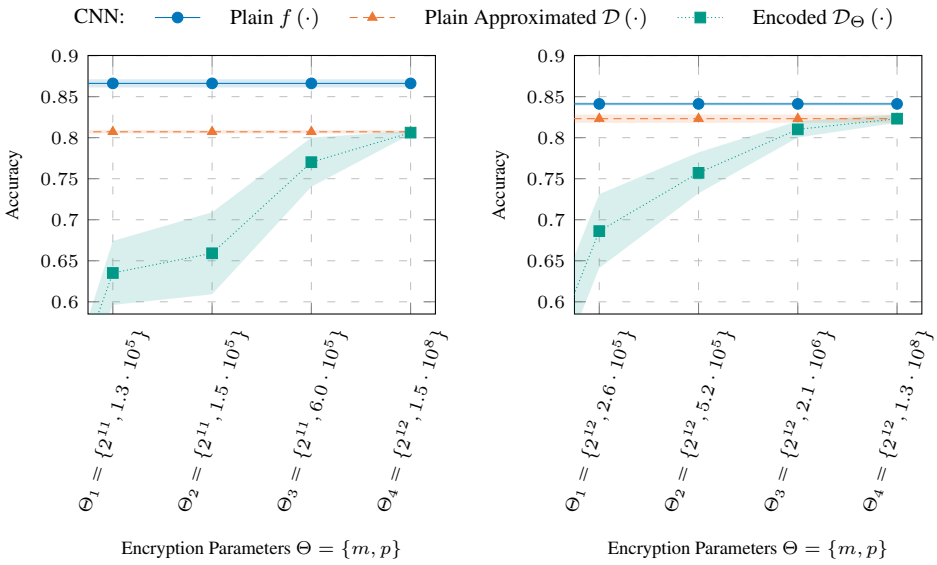
10.2.1 The Provided Convolutional Neural Networks

In this evaluation, the HE-DL architecture provides the following two CNNs, shown in Figure 10.2:

- the *6-layer CNN* (Figure 10.2a) with a convolutional layer with 8 3x3 filters, a 2x2 maximum pooling layer with stride 3, a convolutional layer with 16 3x3 filters and stride 2, a 2x2 maximum pooling layer and two fully-connected layers with 16 and 10 neurons, respectively;
- the *5-layer DLM* (Figure 10.2b) with a convolutional layer with 16 3x3 filters with stride 3 and a ReLU activation function, a 3x3 maximum pooling layer with stride 3 and two fully connected layers with 72 and 10 neurons, respectively.

10.2.2 Datasets

Two datasets have been considered in the analysis:



(a) *The results of the 6-layer CNN.* **(b)** *The results of the 5-layer CNN.*

Figure 10.3: *The recall accuracy results of both the 6-layer CNN and the 5-layer CNN on the FashionMNIST dataset (Xiao et al., 2017), with the standard deviation over five experiments. For each considered encryption parameters Θ_i, three cases are compared: the plain CNN without approximations $f(\cdot)$, the same plain CNN approximated to have only additions and multiplications $\mathcal{D}(\cdot)$, and, finally, the encoded CNN with Θ_i, i.e., $\mathcal{D}_{\Theta_i}(\cdot)$. It is worth noting that with smaller encryption parameters Θ_i (in terms of m and p) than those shown, the accuracy quickly drops to that of a random classifier.*

- the *MNIST* (LeCun, 1998) is a dataset of handwritten digits composed of 70000 grey-scale 28x28 images belonging to 10 classes. From the dataset, 5000 images were used for training and 5000 for validation.

- the *FashionMNIST* (Xiao et al., 2017) is a dataset of fashion products composed of 70000 grey-scale 28x28 images belonging to 10 classes. From the dataset, 60000 images were used for training and 10000 for validation.

In particular, the *FashionMNIST* dataset has been considered in the *recall* modality, while *MNIST* has been used in the *transfer learning* one.

10.2.3 Recall

In *recall* mode, a user wants to use a deep-learning-as-a-service model $\mathcal{D}_\Theta(\cdot)$ published by a Cloud service provider, obtaining the classification y of an input image x. Figures 10.3a and 10.3b show the accuracy of the 6-layers CNN and the 5-layers CNN on the FashionMNIST dataset, respectively, with respect to different values of Θ (the parameter q has been omitted since automatically set). The two CNNs in both the configurations, *plain* and *ap-*

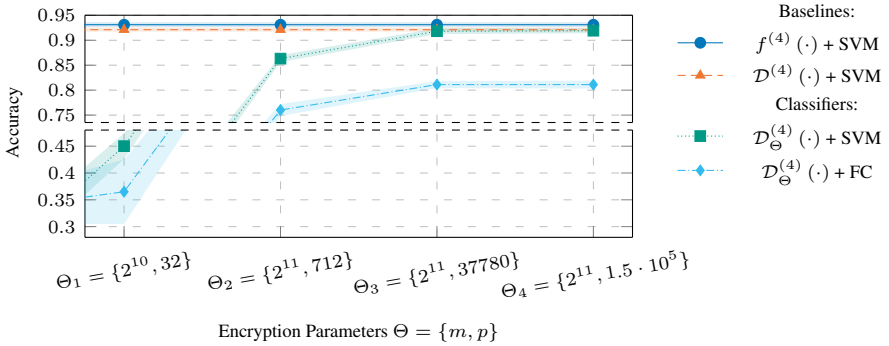


Figure 10.4: *The transfer learning accuracy results on the features extracted at layer $\ell = 4$ of the 6-layer CNN (Figure 10.2a) on the MNIST dataset (LeCun, 1998). For each considered encryption parameters Θ_i, four cases are compared: the plain CNN without approximations $f^{(4)}(\cdot)$ with a SVM-based classifier, the same plain CNN approximated to have only additions and multiplications $\mathcal{D}^{(4)}(\cdot)$ with the SVM, and, the encoded CNN with Θ_i, i.e., $\mathcal{D}^{(4)}_{\Theta_i}(\cdot)$ with either a SVM or a Fully-Connected classifier.*

proximated, have been trained on the FashionMNIST training dataset for 20 epochs, with a learning rate of 0.001.

As expected, the accuracy of the encoded model $\mathcal{D}_{\Theta}(\cdot)$ increases with m and p. In particular, the configuration of parameters Θ_4 (characterized by the largest values of m and p) provides the same performance of the approximated model $\mathcal{D}(\cdot)$ operating on plain data. It is noteworthy to point out that the plain 6-layers CNN (Figure 10.3a) is better than the 5-layers CNN (Figure 10.3b) having higher accuracy than the latter. However, after the approximation, the 5-layers CNN outperforms the 6-layers CNN. This suggests that the approximated CNN $\mathcal{D}(\cdot)$ could be designed from scratch.

10.2.4 Transfer learning

In *transfer learning* mode, the user relies on deep-learning-as-a-service $\mathcal{D}_{\Theta}(\cdot)$ as a feature extractor to train a local classifier, as described in Section 8.3.3. Two types of classifiers have been used, i.e., an SVM-based classifier and a fully-connected-based classifier. Both classifiers have been trained using the features extracted from images coming from the MNIST (LeCun, 1998) dataset, using the first four layers of the pre-trained 6-layers CNN. In particular, 5000 images were used for the training of the classifiers and 5000 for the testing. Figure 10.4 shows the accuracy of the SVM-based and Fully-Connected classifiers. Different values for Θ show the impact on the precision of the extracted features, hence on the accuracy of the trained classifiers. Here, two main comments arise. First, moving from Θ_3 to Θ_4 (with a relevant increase in the parameter p) does not induce a significant improvement in the accuracy. This means that the value $p = 37780$ well characterizes the processing chain of $\mathcal{D}_{\Theta}(\cdot)$. Secondly, Θ_2 for the 6-layers CNN in the recall scenario equals Θ_4 in the transfer learning scenario. However, in the latter case, this set of parameters provides enough NB and precision to carry out the computations correctly, whereas it does not in the former case. This can be explained by the fact that the number

Table 10.5: *The time t required to process an image for each described configuration, with a common PC as a client and an Amazon EC2 instance as a server. The three components of t are t_c, the time required for the local encryption/decryption, t_t, the time for the data transfer, and t_s, the time required for the processing on the Cloud. The proposed values are expressed in seconds.*

		t_c	t_t	t_s	$t = t_c + t_t + t_s$
Recall 6-layer CNN	Θ_1	2.2 ± 0.2	3.7 ± 0.0	11.8 ± 0.1	17.7 ± 0.3
	Θ_2	2.2 ± 0.1	3.7 ± 0.0	11.9 ± 0.1	17.8 ± 0.2
	Θ_3	2.1 ± 0.1	3.7 ± 0.0	11.9 ± 0.0	17.7 ± 0.1
	Θ_4	4.7 ± 0.3	14.7 ± 0.0	49.7 ± 0.5	69.1 ± 0.8
Recall 5-layer CNN	Θ_1	5.2 ± 0.0	14.7 ± 0.0	26.2 ± 0.3	46.1 ± 0.3
	Θ_2	5.2 ± 0.0	14.7 ± 0.0	26.1 ± 0.1	46.0 ± 0.1
	Θ_3	5.2 ± 0.0	14.7 ± 0.0	25.8 ± 0.1	45.7 ± 0.1
	Θ_4	5.2 ± 0.0	14.7 ± 0.0	25.8 ± 0.1	45.7 ± 0.1
Transfer Learning	Θ_1	1.2 ± 0.0	2.0 ± 0.0	5.5 ± 0.0	8.7 ± 0.0
	Θ_2	2.4 ± 0.1	3.9 ± 0.0	11.6 ± 0.1	17.9 ± 0.2
	Θ_3	2.4 ± 0.0	3.9 ± 0.0	11.5 ± 0.0	17.8 ± 0.0
	Θ_4	2.4 ± 0.0	3.9 ± 0.0	11.5 ± 0.0	17.8 ± 0.0

of encoded layers in this transfer learning scenario is lower than in the recall one.

10.2.5 Timing Measurements

To complete the evaluation of the proposed HE-DL architecture, this section details the time measures of the PyCrCNN implementation (Disabato et al., 2020). More in detail, these measures are the computational times on both the client and the server-side and an estimation of the transmission time to exchange information. The settings consider a single image taken from the FashionMNIST dataset for the *recall* modality and from the MNIST dataset for the *transfer learning* modality, in a single-threaded scenario. The models $\mathcal{D}_\Theta(\cdot)$ have been encoded with the same Θ used for the analysis of the accuracy described in Sections 10.2.3 and 10.2.4.

Table 10.5 reports such measures. More in details:

- t_c is the time spent on the client to generate the keys couple (k_p, k_s), to execute the encryption function $E(x, \Theta, k_p)$ and the decryption function $D(\hat{y}, \Theta, k_s)$. The employed client machine used has a 2.30GHz 64-bit dual-core processor and 8 192 MB of RAM.

- t_s is the time spent by the server to encode the model $\mathcal{D}(\cdot)$ and process the encrypted image, $\mathcal{D}_\Theta(\hat{x})$. The considered server is an Amazon EC2 instance with 72 64-bit cores at 3.6GHz and 144 GB of RAM.

- t_t estimates the transmission times of sending the encrypted image $\hat{x}$ and receiving back the encrypted result $\hat{y}$ in an high-bandwidth scenario, where the employed

transmission technology is the *Wi-Fi 4* (standard IEEE 802.11n) using a single-antenna with 64-QAM modulation on the 20 MHz channel with the data-rate $\rho = 72.2Mb/s$ (Xiao, 2005).

Two main comments arise. First, as expected, all the three component of the computational times increase with m. More specifically, t_c and t_s increase due to the larger computational load required to process encrypted data with larger m, while t_t increases due to the increase of the size of the ciphertexts. In addition, t_c is always lower than t_s since $E\left(I, \Theta, k_p\right)$ and $D(\hat{y}, \Theta, k_s)$ are less computational demanding than $\varphi_{\Theta}(\hat{I})$. Second, an increase in p does not result in a variation of the computational times ts. All in all, p should be tuned focusing on the accuracy of the results, while m must be tuned by trading-off accuracy and computational load.

Part V

From the Laboratory to the Wild

CHAPTER 11

From the Laboratory to the Real World

The previous chapters focused on the definition of a methodology as general as possible. In particular, Chapters 5 and 6 design solutions for TML devices with the possibility to learn on the device and thus adapt over time, whereas Chapter 7 exploits a different paradigm by distributing the computation across a group of connected TML devices.

In this Chapter, some of the techniques presented so far have been tailored to two real-world scenarios: detecting the calls of birds in rural or wild environments without the presence of humans and with low connection bandwidth available (Section 11.1); and modeling the highly-non stationary solar activity from the recorded magnetograms (Section 11.2).

11.1 Birdsong Detection in the Wild

In recent decades, bird populations have decreased significantly in many areas around the world (Rosenberg et al., 2019). Furthermore, there is growing evidence that the environmental effects of urban growth and human activities, e.g., increasing noise and night-time light, are at least a partial cause (Injaian et al., 2018; Senzaki et al., 2020). Therefore, it is essential to accurately survey bird activity to better understand the behavior of species and individuals in a variety of habitats, from urban to wildland. While there has been much progress in the detection and classification of bird vocalizations (Priyadarshani et al., 2018), two primary challenges remain. First, discrimination of bird vocalizations from other sounds –especially challenging in suburban/urban environments and near roads– remains a technological challenge. Recently, this has been addressed with sophisticated

signal processing and machine learning algorithms. Secondly, improved algorithms often require more memory and a more significant computational load. Hence, it is more challenging to deploy cost-effective IoT systems for vocalization sensing in the field at scales supporting comprehensive surveys and a more accurate assessment of the distribution of bird species.

There are several excellent algorithmic solutions for birdsong detection and classification, especially those based on deep learning; however, they do not take into account IoT systems' technological constraints. The overarching goal of this application is to bridge this gap by introducing a novel algorithmic solution for birdsong detection based on machine learning that can be executed on off-the-shelf IoT devices.

The remaining of the section is organized as follows. Section 11.1.1 investigates the bird-related literature. Section 11.1.2 proposes the ToucaNet pipeline to detect birdsongs without taking into account the IoT constraints on memory, computation, or energy. Then, Section 11.1.3 describes how to approximate the ToucaNet to take into account IoT technological constraints and presents the BarbNet implementation on an STM32H743ZI microcontroller. Finally, Section 11.1.4 evaluates the two proposed architectures and presents the benchmarks on the considered MCU.

11.1.1 Birdsong Literature

The extensive birdsong detection literature has developed solutions that can be grouped into two main families according to the processing stage in which they operate: preprocessing techniques and detection/recognition techniques.

One of the most used preprocessing approaches consists of transforming the acquired waveforms into spectrograms computed through a complex Short Time Fourier Transform (STFT), often rescaled to the Mel-Scale (Frommolt and Tauchert, 2014; Lasseck, 2013; Neal et al., 2011; Potamitis, 2014; Towsey et al., 2012). Other approaches rely on the extraction of the Mel Frequency Cepstral Coefficients (MFCCs) (Briggs et al., 2009; Dufour et al., 2013; Graciarena et al., 2010; Kogan and Margoliash, 1998; Murcia and Paniagua, 2013), or Discrete Wavelet Transform (DWT)–based features (Bastas et al., 2012). Approaches aiming at directly processing the waveforms are also emerging (Priyadarshani et al., 2020). Several works also bring into play noise reduction techniques to mitigate or remove environmental or anthropogenic noise. These techniques can be applied on the waveform (Priyadarshani et al., 2016) or spectrogram (Lasseck, 2013; Potamitis, 2014). However, the most common noise sources overlap with low-frequency bird calls (potentially masking them) (Potamitis, 2014), hence reducing the effectiveness of subsequent detection techniques (Aide et al., 2013; Fox et al., 2006).

In the field of birdsong detection/recognition, available techniques can be grouped according to the type of approach: signal processing or machine/deep learning.

Techniques following the first approach are, for example, (Lasseck, 2013) and (Potamitis, 2014), where segmentation of spectrograms for birdsong detection is proposed, achieving promising results on clear noise-free calls but having reduced performance when the signal is weak compared with interference and noise. Similarly, (Neal et al., 2011) and (Towsey et al., 2012) attempted to detect time/frequency boxes in spectrograms representing the bird calls. Finally, (Frommolt and Tauchert, 2014) proposed template matching to detect and isolate calls within spectrograms, whereas (Anderson et al., 1996) used dynamic time warping directly on the input waveform.

Machine learning techniques for birdsong detection and recognition have mainly focused on CNNs (Berger et al., 2018; Grill and Schlüter, 2017; Lasseck, 2018; Mukherjee et al., 2018; Ruff et al., 2019). In this setting, (Grill and Schlüter, 2017) suggested two different types of CNN: the former working on the spectrogram of the whole audio input and the latter processing spectrograms computed on shorter audio subsequences with all the provided predictions merged at the end. (Lasseck, 2018) started from a pretrained CNN, either an Inception v3 (Szegedy et al., 2016) or a ResNet (He et al., 2016), and fine-tuned it to the bird detection problem in a transfer-learning fashion (Yosinski et al., 2014). (Vesperini et al., 2018) proposed a capsule-based DL architecture that organizes the convolutional layers in capsules whose outputs are "shared" in such a way that one portion of the spectrogram can leverage the knowledge acquired in other spectrogram locations. Finally, (Himawan et al., 2018) and (Mukherjee et al., 2018) proposed two recurrent DL architectures to detect bird calls in audio signals. These DL-based approaches usually outperform the traditional signal processing techniques (Lasseck, 2018; Ruff et al., 2019). For the recognition task, a few techniques focusing on particular groups of bird species are emerging (Brooker et al., 2020; Ferreira et al., 2020; Koh et al., 2019). (Ruff et al., 2019) proposed a CNN-based architecture to detect and recognize six nocturnal owl species, whereas (Lee et al., 2012) proposed a classification algorithm for 28 bird species based on Gaussian Mixture Models (GMMs). More specifically, this solution models the likelihood of the considered bird species through GMMs, with classification according to the highest likelihood.

All in all, several solutions for birdsong detection are available in the literature, but none of them focus on implementing such techniques on real-world IoT devices. Interestingly, there are DL techniques for audio sources on microcontrollers (Banbury et al., 2021; Disabato and Roveri, 2020), though not for the application scenarios considered in this paper.

11.1.2 ToucaNet: a Pipeline to Detect Bird Calls

The architecture of the proposed ToucaNet bird detector, which is depicted in Figure 11.1b, comprises two main steps:

(i) Acquisition and Preprocessing,

(ii) DL-based Birdsong Detector ψ.

Acquisition and Preprocessing. Let f_a be the microphone's sampling frequency (in Hertz) and t_a be the acquisition time window (in seconds). Let $x \in \mathbb{R}^d$, with $d = f_a \times t_a$, be the d-dimensional vector acquired by the microphone, representing the waveform to be preprocessed and subsequently processed by DL-based birdsong detection.

Once the microphone acquires the waveform x, the preprocessing phase converts it into the corresponding spectrogram $\bar{x}$ computed via the (absolute value of the complex) STFT with n_{fft} bins over constant-length windows spaced by h_l samples, with both n_{fft} and h_l constrained to be powers of two.[19] The outcome of the STFT is a two-dimensional matrix with $\bar{x}_r = 1 + n_{fft}/2$ rows and $\bar{x}_c = 1 + d/h_l$ columns, which is converted into a

[19]The Mel-spectrogram is not considered, since it uses a human-based frequency scale.

three-dimensional image by means of a color-map, i.e., $\hat{x} \in \mathbb{R}^{\bar{x}_r \times \bar{x}_c \times 3}$, to enable transfer learning (see below). In the following, $\hat{x}$ will be called a spectrogram.

The DL-based Birdsong Detector. The DL-based bird detector $\psi(\hat{x}, \theta)$ with parameters θ of the ToucaNet receives as input the spectrogram $\hat{x}$ and produces as output its binary classification $y \in \{0, 1\}$, where $y = 1$ if a bird call is present within the waveform, and $y = 0$ otherwise. This formalization can be extended easily to the case of birdsong recognition by simply considering the supervised information y to belong to a discrete set of classes representing the various bird species. In what follows, to simplify the notation, θ will be omitted from $\psi(x, \theta)$.

Since the spectrogram is interpreted as a colored image $\hat{x}$, it is possible to leverage approaches from the image classification field. More precisely, in the context of this work, the ToucaNet birdsong detector ψ is built upon a ResNet-18 (He et al., 2016), leveraging all its convolutional layers (namely, up to layer *conv5_1*). A 9×14 average pooling filter (with stride 1 and no padding) follows the convolutional structure. Finally, a fully connected classifier labels the 512 extracted features into the desired two classes (Figure 11.1a depicts the ToucaNet architecture in detail).

The convolutional structure of the ToucaNet is initialized with the weights associated to the optimal solution of Resnet-18 on the ImageNet classification task (Deng et al., 2009), hence relying on a transfer learning approach (Yosinski et al., 2014) to speed up the training phase. The fully-connected layer is instead initialized with a uniform distribution in the interval $\left(-1/\sqrt{fc_{in}}, 1/\sqrt{fc_{in}}\right)$, where fc_{in} is the input size of the layer itself, i.e., 512.

To train ψ, an n-dimensional data set $D = \{(x_i, y_i)\}_{i=1}^{n}$ is used and the parameters θ are optimized as

$$\arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} l\left(\hat{x}_i, y_i, \theta\right), \tag{11.1}$$

where $l(\cdot)$ is a classification loss function and $\hat{x}_i$ is the spectrogram of the waveform x_i. ψ is trained for 14 epochs with stochastic gradient descent, momentum 0.9, and a learning rate of $5 \cdot 10^{-2}$, decreased by a factor of 10 after the eighth and the twelfth epochs.

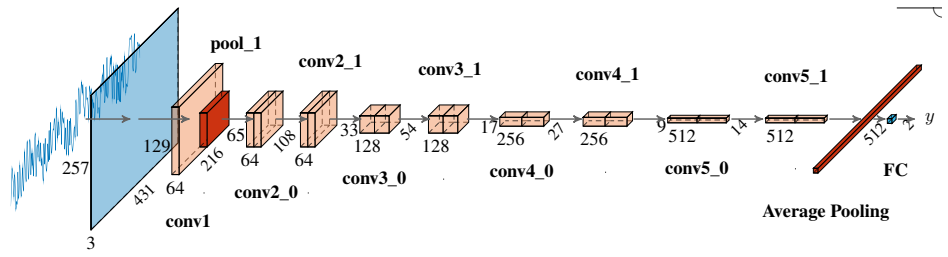
Since our final goal is to move our solution to an IoT unit, the memory footprint m_ψ and the computational load c_ψ of ψ have been carefully evaluated. Following Eq. (2.2) and Eq. (2.3), m_ψ and c_ψ are defined as

$$m_\psi = |\theta| \cdot m_p + m_{in}, \tag{11.2}$$

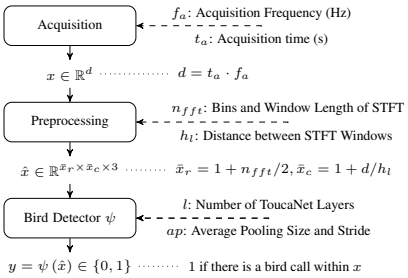
$$c_\psi = n_{mul}, \tag{11.3}$$

where $|\theta|$ is the cardinality of θ, n_{mul} is the number of multiplications required to compute the output, m_{in} is the memory needed to store the input x (and all its transformations), and m_p is the memory required to store a single parameter (typically a 32-bit floating-point data type).

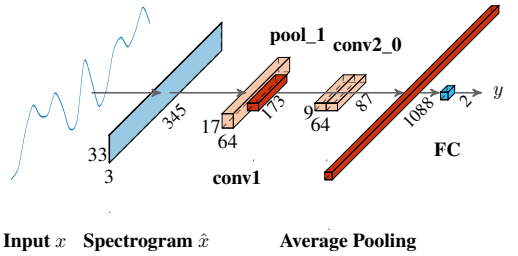
As shown in Figure 11.3, the ToucaNet has detection capabilities in line with the best solution available in the related literature given the same acquisition frequency ($f_a = 22050$ Hz) but half its memory footprint ($m_\psi = 44.714$ MB) and about 80% of its computational demand ($c_\psi = 4.255$ billion multiplications).



(a) *The architecture of the ToucaNet Bird Detector ψ, that receives the spectrogram $\hat{x}$ (computed with $n_{fft} = 512$, $h_l = 512$) of a waveform x sampled at $f_a = 22050$ Hz for $t_a = 10$s, and provides its output classification y.*



(b) *A scheme of the pipeline with the parameters that can be approximated.*



(c) *The BarbNet Bird Detector ψ architecture, that receives the spectrogram $\hat{x}$ ($n_{fft} = 64$ and $h_l = 64$) of a waveform x sampled at $f_a = 2205$ Hz for $t_a = 10$s, and provides its output classification y on a STM32H743ZI IoT device.*

Figure 11.1: *Comprehensive scheme of the proposed solution to detect bird calls in audio acquired on the field.*

These computational demand and memory footprint improvements provide a more efficient but not less effective birdsong detection solution. However, these requirements are still too demanding to allow deployment on IoT units. Therefore, in Section 11.1.3, the proposed ToucaNet is approximated to enable its deployment on off-the-shelf IoT systems.

11.1.3 BarbNet: the Approximated ToucaNet for IoT Units

To deploy a bird song detector at the IoT edge, its memory and computational requirements must meet the constraints ($\bar{m}$ and $\bar{c}$) of the selected IoT node. To achieve this, ToucaNet has been revised by introducing approximations both at the acquisition-preprocessing and birdsong detection layers. Moreover, reducing the computational demand as much as possible also saves energy and prolongs the IoT device activity in the field. Figure 11.1b depicts all the parameters that can be approximated in the ToucaNet architecture. Following this idea, the BarbNet, i.e., the approximated version of ToucaNet that can satisfy the memory and computational constraints of the STM32H743ZI MCU board, is defined.

Approximating the Input. The Acquisition and Preprocessing step of the ToucaNet comprises the STFT to compute the spectrogram $\hat{x}$, whose computational complexity is $O\left(d \cdot n_{fft} \cdot \log\left(n_{fft}\right)\right)$, which is negligible compared to the one required by the DL bird detector. However, the total memory footprint m_{in} of this step is non-negligible and comprises the memory footprint m_x of the input signal x and the memory footprint $m_{\hat{x}}$ of the spectrogram $\hat{x}$, i.e.,

$$m_{in} = (m_x + m_{\hat{x}}) \cdot m_p = (d + \bar{x}_r \cdot \bar{x}_c \cdot 3) \cdot m_p. \tag{11.4}$$

Therefore, the memory footprint m_{in} of the Acquisition and Preprocessing step depends on the sampling frequency f_a, the acquisition time t_a, and the STFT parameters n_{fft} and h_l. Consequently, although f_a is application-dependent and, in principle, must be set two times larger than the maximum frequency in the waveforms (following the Nyquist-Shannon sampling theorem), in this context, it needs to be considered as an approximation parameter. Moreover, reducing f_a will also reduce the computational demand c_ψ of the bird detector since the spectrogram will be smaller. Since reducing f_a might introduce aliasing, an anti-aliasing filter (tailored to f_a) is advised before the ADC to limit the input signal bandwidth.

For the other three parameters, t_a, n_{fft}, and h_l, the former is set according to the available data sets (i.e., fixed to $t_a = 10$s), and the latter two are set such that each window of the spectrogram is 30ms and non-overlapping (refer to Table 11.1 for details) to cover approximately one syllable of a bird call (Hart et al., 2018; Marler and Isaac, 1960; Paliwal et al., 2010).

Approximating the DL-Based Birdsong Detector. The memory footprint and computational demand of the ToucaNet birdsong detector are controlled by reducing the number of parameters θ and the number of features in the input to the ToucaNet FC-layer. In the former approach, the ToucaNet birdsong detector is approximated by removing layers from the ResNet-18-based part of the pipeline. The higher the number of layers ℓ kept in the approximated ToucaNet, the higher the detection capability at the expense of a larger memory footprint and more significant computational load. Section 11.1.4 analyses all the configurations using the first five convolutional blocks, i.e.,

$$\ell \in \{pool1, conv2\_0, conv2\_1, conv3\_0, conv3\_1\}. \tag{11.5}$$

The choice of ℓ strictly depends on the IoT unit constraints $\bar{m}$ and $\bar{c}$. For instance, the memory constraint $\bar{m}$ of the STM32H743ZI unit requires the BarbNet (Figure 11.1c) to have $\ell = conv2\_0$.

To reduce the number of features in input to the final classifier, a dimensionality reduction operator consisting of an $ap \cdot ap$ average pooling layer (stride ap and no padding) is introduced, where ap is an additional hyper-parameter of the model ψ.[20] Although other dimensionality reduction operators could have been considered, such as PCA (Pearson, 1901) or autoencoders, the average pooling operator has the advantage of having negligible computational demand and no memory footprint, making it a very appealing choice in this setting.

[20]Please note that setting $ap = 1$ skips this operator.

Table 11.1: *A summary of the considered acquisition frequencies f_a, along with the details of the resulting spectrograms and their memory occupation $M_{\hat{x}}$ with a 32-bit data type.*

f_a (Hz)	n_{fft}	h_l	$\bar{x}_r$	$\bar{x}_c$	$M_{\hat{x}}$ (KB)
1100	32	32	17	344	68.53
2205	64	64	33	345	133.42
4410	128	128	65	345	262.79
8820	256	256	129	345	521.54
17640	512	512	257	345	1039.04
22050	512	512	257	431	1298.05

Table 11.2: *The detailed memory footprint (with a 32-bit data type) and the computational requirements of the BarbNet implemented on the STM32H743ZI. To optimize the memory, two arrays only are used to store the activations (an asterisk marks the activations re-using such arrays).*

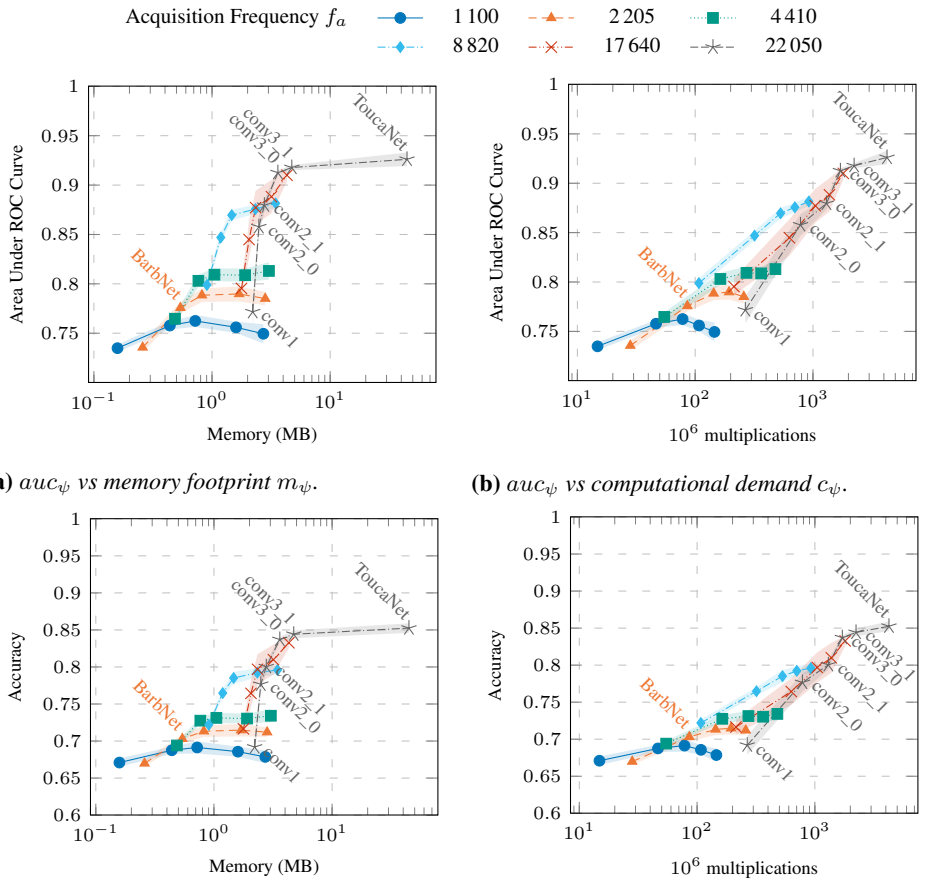
	Memory Footprint (KB)	10^6 operations
Audio ($t_a = 10$ s, $f_a = 2205$ Hz)	*172.76	-
Spectrogram ($n_{fft} = 64, h_l = 64$)	*133.42	-
Conv1 (Weights)	37.75	-
Pool1 (Weights)	-	-
Conv1–Pool1 (Activations)	195.75	28.120
Conv2_00 (Weights)	144.00	-
Conv2_00 (Activations)	195.75	28.865
Conv2_01 (Weights)	144.00	-
Conv2_01 (Activations)	*195.75	28.865
Avg Pool with $ap = 5$ (Weights)	-	-
Avg Pool with $ap = 5$ (Activations)	*4.25	0.010
FC Classifier (Weights)	8.5	-
FC Classifier (Activations)	0.008	0.002
Convolutions Auxiliary Memory	28.50	-
Total	754.26	85.878

A Bird Song Detector on an ARM Cortex-M7: BarbNet In this work, the target IoT device is the STM32 Nucleo H743ZI2 MCU endowed with a 480 MHz ARM Cortex-M7 processor, 1024 KB of RAM, 2 MB of Flash, and no Operating System. The memory constraint $\bar{m}$ is set to the RAM size, i.e., $\bar{m} = 1024$ KB. The BarbNet (Figure 11.1c) is the approximated version of the ToucaNet satisfying these constraints.[21]

Table 11.2 details the memory and computational demand of BarbNet, assuming a 32-bit floating-point representation for all the weights and activations. More specifically, the BarbNet samples at $f_a = 2205$ Hz and generates spectrograms $\hat{x}$ with $n_{fft} = 64$ and $h_l = 64$. In principle, this small value for f_a might prevent the detection of higher-frequency birdsongs but Section 11.1.4–Figures 11.2 and 11.3 show that the figures of merit are still good on real benchmarks.

The required memory footprint for both the signal of size $d = 22050$ and the resulting spectrogram ($\bar{x}_r = 33$, $\bar{x}_c = 345$) is 306.18 KB. The first convolutional layer of the ResNet-18, characterized by 64 7×7 filters with stride 2 and padding 3, processes the computed spectrogram $\hat{x}$. A batch-normalization layer and a 3×3 maximum pooling with

[21] Please refer to Section 11.1.4 for discussion of other feasible configurations.



(a) auc_ψ *vs memory footprint* m_ψ.

(b) auc_ψ *vs computational demand* c_ψ.

(c) acc_ψ *vs memory footprint* m_ψ.

(d) acc_ψ *vs computational demand* m_ψ.

Figure 11.2: *Outcomes in terms of AUC (a,b) and accuracy (c,d) against the memory footprint and computational demand by the ToucaNet and its approximations $\mathcal{A}$ in terms of acquisition frequency f_a and layer ℓ. The layers ℓ are annotated for one plot only. The memory footprint and the computational demand are defined in Eqs.* (11.2) *and* (11.3).

stride 2 follow. This block accounts for 28 million operations, requires 37.75 KB of memory to store the weights, and generates $9 \cdot 87 \cdot 64$ activations, requiring further 195.75KB of memory. Two 3×3 convolutions compose the second convolutional block (*conv_0*) with 64 filters with stride 1 and padding 1, each followed by a batch normalization. This block does not change the activation size, has a memory footprint of 288KB, and requires nearly 58 million operations. Finally, the average pooling layer has (size and stride) $ap = 5$ and requires 10000 multiplications. The pooling generates, after flattening, 1088 activations that are provided to the fully connected classifier, which in 2176 operations yields the final classification label, requiring 8.5KB of memory. Consequently, the total number of Barb-

Net multiplications is 84 million, and its total memory footprint is 755KB.[22] Note that two arrays are used alternatively to store all the inputs and intermediate representations, saving 506.18KB of memory.

To further reduce the computational demand, a few device-dependent optimizations have been implemented in the BarbNet deployed version on the STM32H743ZI unit, relying on the CMSIS-DSP Cortex-M7 processor's instruction set:

- As in the most prominent DL libraries (e.g. (Chetlur et al., 2014)), the convolution layer has been converted into a matrix multiplication. The convolutional input patches (i.e., all the activation pieces the convolutional filters are multiplied by) are unrolled and organized as rows in the first matrix of shape $(n_{patches}, f_s)$, with $n_{patches}$ the number of patches and f_s the dimension of each (unrolled) filter. The n_{fft} convolutional filters (of size f_s) are instead rolled out and arranged as columns in the second matrix of shape (f_s, n_{fft}). Finally, the multiplication result of size $(n_{patches}, n_{fft})$ between the *patches* and filter matrices is reshaped to match the convolutional output;

- Since convolutional filters embrace overlapping patches, the previously described creation of the *patches* matrix will result in a non-negligible increment of the memory footprint. To balance this increment, the maximum number of patches multiplied at any given time, $\hat{n}_{patches}$, is defined as an additional parameter of the problem. In our implementation, $\hat{n}_{patches}$ is fixed to the number of columns, which means that to carry out a convolution, the number of employed matrix multiplications equals the number of convolutional input rows;

- Batch normalization and maximum pooling operations are computed simultaneously with the previous convolutional layer;

- The zero-padding (to be done in all the convolutional and pooling layers) is implemented without explicitly instantiating the "padded" array;

- The fully-connected layer is carried out as a matrix multiplication via CMSIS-DSP instructions.

11.1.4 Evaluating the ToucaNet and the BarbNet

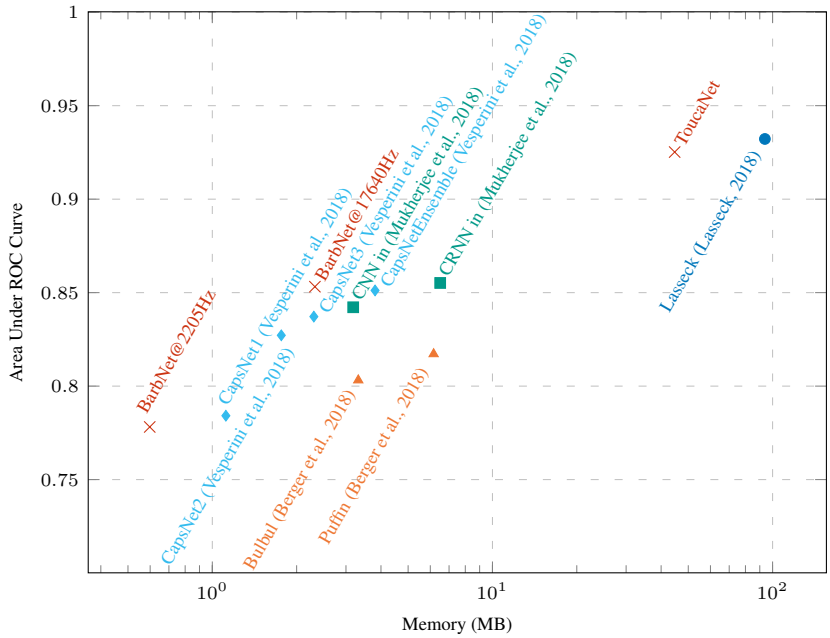
This section details the birdsong recognition experimental results by evaluating the proposed ToucaNet pipeline and its several possible approximations, including the BarbNet, with the solutions available in the related literature. Finally, this Section benchmarks the BarbNet implementation on the STM32H7 IoT device in terms of execution time, power consumption, and system lifetime.

Data Sets and Figures of Merit. The experimental results have been collected on three different datasets (Stowell et al., 2019): *Warblr*, composed of approximately 8000 10s recordings from smartphones (in UK); *Free-Field*, containing 7690 10s recordings from a research project; and *BirdVox-DCASE-20k*, namely the *remote monitoring flight calls*

[22] The total number of multiplications is an upper bound since it does not consider all the optimizations.

Solution	Input Details			m_ψ (MB)		c_ψ	auc_ψ
	f_a (Hz)	t_a (s)	Dimensions	Input	Parameters	10^9 ops	
BarbNet	2205	10	33x345x3	0.215	0.383	0.086	0.778 ± 0.006
BarbNet (2)	17640	10	257x345x3	1.941	0.383	0.631	0.853 ± 0.018
ToucaNet	22050	10	257x431x3	2.108	42.606	4.255	0.925 ± 0.008
Lasseck (Lasseck, 2018)	22050	10	299x299x3	1.864	≈ 92	≈ 5	0.932
CRNN	44100	10	128x768	2.057	4.452	> 6.460	0.855
CNN (Mukherjee et al., 2018)	44100	10	128x768	2.057	1.127	6.460	0.842
Puffin	44100	10	80x1000	1.897	≈ 4.270	≈ 0.162	0.817
Bulbul (Berger et al., 2018)	44100	10	80x1000	1.897	1.423	0.054	0.803
CapsNet1	16000	10	501x40	0.687	0.431	-	0.784
CapsNet2	16000	10	501x40	0.687	1.076	-	0.827
CapsNet3	16000	10	501x40	0.687	1.617	-	0.837
CapsNet Ensemble (Vesperini et al., 2018)	16000	10	501x40	0.687	3.124	-	0.851

(a) *Details of the birdsong detection state of the art solutions (each group corresponds to the same authors).*



(b) *auc_ψ vs memory footprint m_ψ outcomes obtained by the BarbNet working on an STM32H743ZI (with $\bar{m} = 1024$ KB) and other solutions available in the related literature.*

Figure 11.3: *A comparison of the ToucaNet and BarbNet with the related literature. The memory footprint and the computational demand of each solution is computed according to Eqs.* (11.2) *and* (11.3) *and the description of the solution in the corresponding paper. Since the employed datasets are the same, the figure of merit is reported as it is.*

dataset, that contains 20000 10s audio clips recorded remotely near Ithaca, NY, USA, during autumn 2015.

The acquisition time is fixed to $t_a = 10$s to use the available supervised information and compare it with the related literature on the same data sets. Other data sets in the field, such as the *Chernobyl Exclusion Zone (CEZ)* or the *Remote monitoring night-flight calls in Poland*, have not been considered since the labels are not publicly available.

In the comparison of the birdsong detection solutions, two figures of merit have been employed: the birdsong detection accuracy acc_ψ and the Area Under Curve (AUC) auc_ψ of the Receiver Operating Characteristic (ROC) on the class *bird*. In the experimental results, such figures of merit have been computed through 10-fold cross-validation.

Pareto Front of the ToucaNet and its Approximations. As described in Section 11.1.3, both the sampling frequency f_a and the DL-based detector ψ need to be tailored to the target IoT device. Therefore, in this section, the ToucaNet and a set of its approximations are analysed in terms of acc_ψ, auc_ψ, m_ψ, and c_ψ. The considered set of approximations is

$$\mathcal{A} = \{(f_a, \ell) : f_a \in \{1\,100, 2\,205, 4\,410, 8\,820, 17\,640, 22\,050\}\ \text{Hz}, \\ \ell \in \{conv1, conv2\_0, conv2\_1, conv3\_0, conv3\_1\}\}, \quad (11.6)$$

where ℓ defines the ResNet block's considered number of convolutional layers within the ToucaNet (see Figure 11.1a), so if ℓ is $conv2\_1$ then the ResNet-based detector is approximated by considering all the layers up to $conv2\_1$. It is crucial to point out that the BarbNet belongs to the set $\mathcal{A}$, with $f_a = 2205$ Hz and $\ell = conv2\_0$.

All the ToucaNet approximations are trained similarly to what is described in Section 11.1.2. The initialization of layers follows the same as ToucaNet; the training epochs are 14, the momentum is 0.9, and the learning rate is $5 \cdot 10^{-3}$ decreased by a factor of 10 after the sixth and the tenth epoch.

For each configuration in $\mathcal{A}$, Figure 11.2 reports auc_ψ vs. m_ψ, auc_ψ vs. c_ψ, acc_ψ vs. m_ψ, and acc_ψ vs. c_ψ. In particular, in Figures 11.2a and 11.2b it is possible to observe the Pareto Front generated by the considered ToucaNet approximations. This frontier can be leveraged as a criterion to choose the architecture to deploy according to the constraints imposed by a given IoT device. More specifically, in our case, given the memory constraint $\bar{m} = 1024$ KB of the STM32H743ZI unit, there are five solutions on the Pareto Front that are suitable for the deployment. The BarbNet is the fourth one with the highest AUC. The fifth and best one is unfeasible when considering the intermediate activations of the convolutional layers. Thus it is not considered in the following.

Comparing ToucaNet and BarbNet with the State-of-the-Art Solutions. Figure 11.3 compare both the ToucaNet and the BarbNet with state-of-the-art solutions in the birdsong detection literature (Lasseck, 2018; Mukherjee et al., 2018; Berger et al., 2018; Vesperini et al., 2018). Two main comments arise. First, ToucaNet has detection capabilities in line with the best solution in the literature, i.e., Lasseck (Lasseck, 2018), but with half its memory footprint and about 80% of its computational demand. Second, BarbNet is the only solution satisfying the requirements of the STM32H743ZI unit and, when it samples the input signal x at $f_a = 17640$ Hz, equals or overcomes the accuracy and AUC of all the other solutions with similar memory footprint (Figure 11.3b).

Table 11.3: *The BarbNet experimental execution timings on the STM32H743ZI, measured with an oscilloscope.*

	Time (ms)
Audio ($t_a = 10$ s, $f_a = 2205$ Hz)	10 000.00
Spectrogram ($n_{fft} = 64$, $h_l = 64$)	22.80
Conv1–Pool1	432.00
Conv2_00	1 360.00
Conv2_01	1 470.00
Avg Pool ($ap = 5$)	$\ll 1.00$
FC Classifier	$\ll 1.00$
Total	3 284.80

Table 11.4: *The energy analysis of the BarbNet deployment on the STM32H743ZI, when considering a 3.3V power supply in the "*acquire-classify-sleep*" approach.*

(a) *Energy consumption measurements.*

	t (s)	i (μA)	P (mW)	E (J)
Acquisition	10.000	721.95	2.38	0.000052
Computation	3.285	55 000.00	181.50	0.596228
Sleep	6.715	1.95	0.006	0.000043

(b) *System Lifetime with different Li-Ion batteries (their available energy is 75% of the nominal capacity).*

Battery Capacity	Battery Energy	Lifetime	
mAh	J	Hours	Days
100	1 332	9.31	0.39
250	3 330	23.27	0.97
500	6 660	46.54	1.94
1 000	13 320	93.07	3.88
1 200	15 984	111.68	4.65
1 500	19 980	139.61	5.82
2 000	26 640	186.14	7.76
2 500	33 300	232.68	9.69
3 200	42 624	297.83	12.41

BarbNet on the STM32H7: Execution Time, Energy Consumption, and Lifetime. A detailed experimental analysis to quantitatively measure the execution time, the energy consumption, and the expected lifetime of the BarbNet running on STM32H7 has been conducted, as detailed in the sequel.

Table 11.3 details the execution time results from audio acquisition to the last layer of BarbNet and shows that BarbNet requires 3.285s to compute the classification y, significantly below the time needed to acquire the audio signal (10s). Hence, embedded systems endowed with multi-cores or DMA mechanisms can made the waveform acquisition and classification simultaneously. In the technological scenario with a single core and DMA not considered, we suggest two STM32H7 running in opposite modes. While the former is acquiring the waveform, the latter predicts (with the BarbNet) on the previously acquired waveform and then sleeps up to the subsequent acquisition step. The lifetime evaluation is

based on this "*acquire-classify-sleep*" approach.

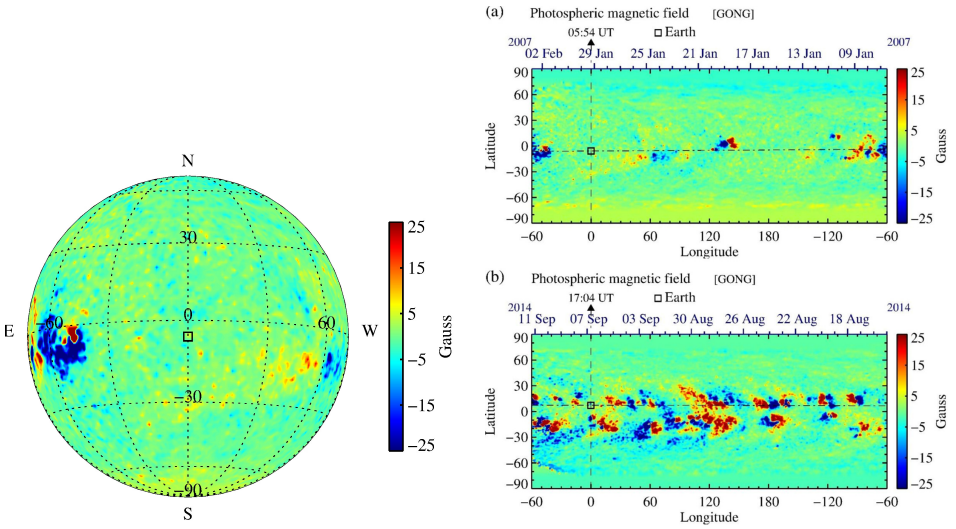
In more detail, Table 11.4a reports the energy required by each step, showing that the most expensive one is the DL-Based Birdsong Detector *computation* step. Given the energy consumptions detailed in Table 11.4a, Table 11.4b analyses the lifetime of the proposed solution on the STM32H743ZI microcontroller with several real Li-Ion batteries. To account for battery non-idealities and degradation over time, only 75% of the nominal battery capacity is considered. Interestingly, the considered IoT unit running the BarbNet provides a 7 days and 12.4 days lifetime when using a 2000 mAh and 3200 mAh capacity Li-ion battery, respectively.

11.2 Solar Magnetograms Activity Identification

Space Weather (SW) represents a chain of processes that begin at the Sun. They can adversely affect technology on Earth and its space environment posing hazards to modern society. For this reason, significant research efforts are undertaken to understand and forecast SW and its impacts on Earth activities. More specifically, the complex interplay between the plasma and magnetic field is the source of solar disturbances such as solar flares and Coronal Mass Ejections (CME). Since the magnetic field controls the solar ejections, the research and (most of the) operational SW models widely use information about the magnetic field. Although regular space and ground-based measurements of the Sun magnetic field are available, they are restricted to the side of the Sun facing the Earth (near side) and to the photospheric/chromospheric heights (Petrie, 2015). At these heights, close to the Sun surface, the plasma density is high enough for a favorable signal-to-noise ratio. The noise level represents a significant challenge for measurements on the limbs of the solar disc, which is the visible surface of the Sun. Furthermore, due to the approximately 7.25-degree tilt of the Sun rotation axis with respect to the ecliptic plane, the polar regions are not always visible, and different techniques are used to fill the gaps in the polar regions (Sun et al., 2011).

SW has many unanswered questions about physical processes and significant gaps still exist. Since a significant amount of solar observations exists in the solar and SW community, Machine Learning (ML) could be a viable solution to provide significant scientific and application advances. Such ML techniques aim at examining the behavior and determining the evolution of the Sun magnetic field. Since measurements of the field are restricted to the near side, there is an interest to predict active regions on the far side of the Sun. This is particularly important for SW forecasting since, as the Sun rotates, these regions emerge on the near side and, thus, the associated solar disturbances are more geoeffective.

This section uses Machine and Deep Learning techniques from Computational Intelligence (CI) to study and characterize the behavior of the Sun magnetic field, whose data come from the Global Oscillation Network Group (GONG) observations (Hill, 2018). GONG provides solar magnetograms of the photospheric magnetic field with 24 hours coverage of the Sun. These data are widely used in the research and operational SW community. For example, GONG synoptic maps are used with numerical models to derive the coronal magnetic field (Nikolić, 2017; Nikolić, 2019) and to forecast CME propagation and arrival times (Steenburgh et al., 2013; Pomoell and Poedts, 2018).



(a) *Photospheric magnetic field of the Sun as viewed from the Earth on January 29, 2007 (05:54 UT). The field is saturated at* ± 25 *Gauss. E, W, N, and S denote East, West, North, and South, respectively. The small square in the image represents the projection of the Earth on the solar disc.*

(b) *GONG synoptic maps of the photospheric magnetic field for (a) January 29, 2007 (05:54 UT) and (b) September 06, 2014 (17:04 UT). The magnetic field is saturated at* ± 25 *Gauss. The central meridian and sub-Earth locations are denoted with dash-dash and dash-dot line, respectively.*

Figure 11.4: *Two examples of magnetic field in the adopted representations.*

11.2.1 Solar Magnetograms, Synoptic Maps, and Data Preprocessing

A solar magnetogram represents an image of the magnetic field on the solar disc. As an example, Figure 11.4a shows the GONG solar magnetogram of the photospheric magnetic field for January 29, 2007 (05:54 UT). The red and blue colors represent the magnetic field directed away from and towards the Sun, respectively. The field in Figure 11.4a represents the radial component of the magnetic field $B_r(R_0, \vartheta, \phi)$, where, R_0 is the radius of the Sun, θ is the colatitude, and ϕ is the longitude. In the figure, a region with a strong magnetic field is visible on the East limb of the solar disc. As the Sun rotates, with about a 27.27 day rotation period, this active region will move from East (E) to West (W) and cross the central meridian. The solar magnetograms can be combined into so-called synoptic maps that represent full-surface maps of the Sun magnetic field and are often used in SW research and operations (Hill, 2018).

More in details, the GONG synoptic maps S are re-meshed and organized as a matrix $[s_{\vartheta,\phi} = B_r\,(R_0, \vartheta, \phi)]$, with uniform 1° resolution in ϑ and ϕ. These re-meshed maps are proven to improve the accuracy of the widely used potential field source surface model of the solar corona and, particularly, of spherical harmonics in the polar regions (Nikolić, 2017; Tóth et al., 2011).

Figure 11.4b shows two examples of re-meshed synoptic maps S, for (a) January 29, 2007 (05:54 UT), and (b) September 6, 2014 (17:04 UT). The map time is associated with

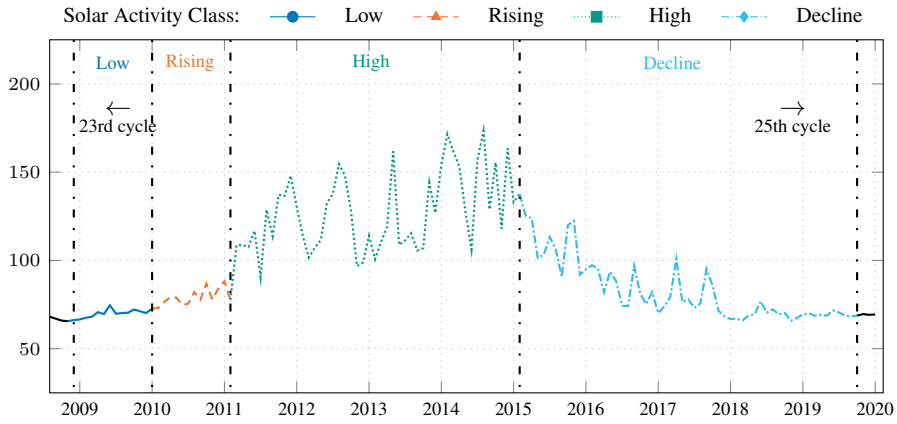


Figure 11.5: *The definition of the classes according to the solar activity on the adjusted 10.7cm solar flux.*

the central meridian that is denoted with the dash-dash line, whereas the dash-dot line denotes projections of the Earth on the map at different times. As the Sun rotates, the region $\phi < -60°$ will cross the central meridian. The synoptic map (a) in Figure 11.4b corresponds to the solar magnetogram from Figure 11.4a and illustrates the magnetic field of the Sun close to the solar cycle 23 minima, whereas Figure 11.4b(b) shows the field around the maximum of solar cycle 24.

The GONG synoptic maps are assembled from many measurements of the photospheric magnetic field, including measurements from different observatories to provide daily coverage of the Sun. All those measurements are weighted when computing the synoptic map. Since a particular measurement can capture only the field on the near side of the Sun, the far side on the synoptic map consists of non-updated past data. Furthermore, due to the limb noise, only the longitudinal region $-60° < \phi < -60°$ most accurately captures the Sun's magnetic field.

Summing up, this application relies on synoptic maps $S \in \mathbb{R}^{\Theta \times \Phi}$, where $\Theta = 360$ and $\Phi = 181$. More in detail, there are two maps per day, from November 1, 2006, to September 30, 2019, typically acquired at 05:04 and 17:04 UT (when the maps are not available at those times, the closest ones are used if the time difference is less than two hours). However, there are some days when no map is available in the GONG archive. The considered period covers the whole solar cycle 24.

11.2.2 Solar Activity Classes Definition

The solar activity within solar cycle 24 has been divided into four (unbalanced) classes, trying to respect as much as possible the physical structure of the problem, starting from what described by Schwabe (Schwabe and Schwabe, 1844):

- *Low* (794 magnetograms): the initial period of a solar cycle, characterized by low solar activity. In our cases, the *low* activity period started at the beginning of cycle 24, estimated in December 2008 up to the end of 2009;

- *Rising* (848): after the *low* activity phase, a small transition period to the *high* activity one exists, here considered as a standalone class. It covers the period from January 1, 2010, to the end of February 2011;

- *High* (2866): this is the phase of the solar cycle characterized by the highest solar activity and the highest solar disturbances. It covers the period from March 1, 2011, to January 31, 2015;

- *Decline* (3406): after the *high* activity phase, the solar activity declines up to the next solar cycle. This period covers the magnetograms from February 2015 to the last available ones in the series.[23]

It is noteworthy to point out that the crisp boundaries between these phases are artificially introduced, while in real-world situations they are smooth. For these reasons, classification errors close to such boundaries are reasonably expected (since most classification algorithms assume sharp boundaries). The definition of classes is shown in Figure 11.5, together with the 10.7 cm solar radio flux that is an indicator of solar activity (Livingston et al., 2012; Valdés et al., 2019).

11.2.3 Solar Activity Modeling through Deep Learning Solutions

The preprocessed solar magnetograms S are converted to images by assigning a three-dimensional color (from any perceptual uniform colormap) in the interval $[-100, 100]$. All the values out of this interval are considered as the interval nearest value, i.e., -100 for negative values and 100 for positive ones. Figure 11.6 presents the proposed Deep Learning-based architecture to address the problem of modeling the solar activity (in the spectrograms). In particular, the solution comprises the following steps:

- a pre-trained Convolutional Neural Network (CNN) $\varrho(\cdot)$ acting as a feature extractor following a *transfer learning* approach (LeCun and Bengio, 1995; Yosinski et al., 2014).

- a dimensionality reduction operator $\varsigma(\cdot)$. This step is crucial to mitigate the *curse of dimensionality* problem deriving from high-dimensional spaces that typically characterize CNNs (Zimek et al., 2012). The considered ς is a Principal Component Analysis (Wold et al., 1987). The feature space defined by $\varsigma \circ \varrho(\cdot)$ represents solar magnetograms S in the reduced space of CNNs.

- Finally, the reduced feature vector can be processed by both supervised and unsupervised ML techniques: Support Vector Machines (Cortes and Vapnik, 1995) as supervised, and k-Means (Lloyd, 1982) and Gaussian Mixture Models as unsupervised clustering techniques. More specifically, the supervised classification is meant to recognize the solar activity classes defined in Section 11.2.2, whereas clustering techniques should validate the definition of these classes by identifying clusters that match them (ideally, each cluster should contain samples belonging to only one class).

[23] The end of solar cycle 24 is indeed estimated to be in late 2019.

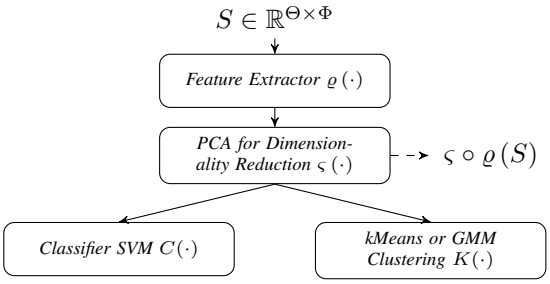


Figure 11.6: *The proposed deep learning architecture to characterize solar magnetograms Ss. On the extracted features $\varsigma \circ \varrho(S)$, various supervised or unsupervised techniques can be applied.*

It is crucial to point out that the proposed deep learning architecture does not require training, except for the computation of the PCA ς. However, nothing avoids refining the feature extractor ϱ, for example, with some solar magnetogram images Ss.

11.2.4 Analysis

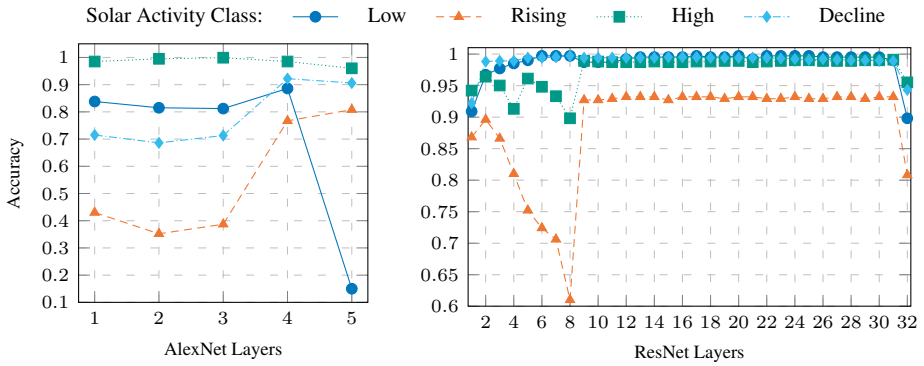
Implementation Details. The architecture, detailed in Section 11.2.3 and shown in Figure 11.6, has been implemented in Python (through the PyTorch (Paszke et al., 2019) framework). In particular, two DLMs have been considered for the feature extractor ϱ: the AlexNet (Krizhevsky et al., 2012) at the end of each convolutional block (precisely, the pooling layer at the first, second, and fifth block, the ReLU non-linearities at the others), and the ResNet-101 (He et al., 2016) after each of the 32 residual blocks. The PCA ς is computed, in each case, on the number of components that keep 95% of the variance.

Analysis Results. Figure 11.7 shows the classification results for both the considered CNNs, with a 50-50% training/testing split of the solar magnetograms.

The AlexNet (Figure 11.7a) shows extremely high accuracy on the *High* activity class as well as some difficulties with the other classes, especially the *Rising* one. The fourth layer (*ReLU4*) exhibits the best classification accuracy on all classes with a global accuracy of 0.925. In contrast, the fifth layer (*pool5*) has a very low accuracy on the *Low* class, despite comparable or better accuracies on other classes w.r.t. *ReLU4* layer.

Figure 11.7b shows the ResNet classification results. The first eight layers have a low accuracy on the Low class, whereas the last layer has a small accuracy for all the classes. However, layers 9 to 31 (at the end of fourth and fifth convolutional layers) are able to provide high accuracy, i.e., higher than 0.9 on all classes. These results indicate that a nine-layer architecture is appropriate. Consequently, this specific architecture will be considered in the sequel.

Figure 11.8 shows the distribution of the magnetograms on the feature space of the considered architecture, with a PCA aiming at keeping the 95% of the variance as a dimensionality reduction operator. Interestingly, Figure 11.8a exhibits a clearly distinctive structure of the years, with only a slight overlapping between neighboring ones. From the beginning of the magnetogram series ($\leq$ 2009, top-left, beginning of the 24th solar



(a) *The results of the SVM on top of AlexNet + PCA feature extractor.*

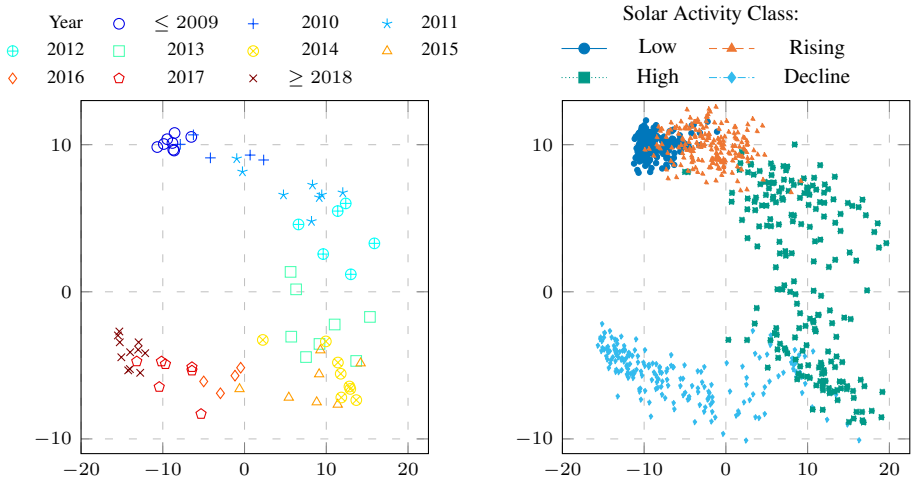
(b) *The results of the SVM on top of ResNet + PCA feature extractor.*

Figure 11.7: *The accuracy of the four defined solar activity's classes on different feature spaces. Each feature space has a feature extractor either the AlexNet (Figure 11.7a) or the ResNet (Figure 11.7b) and as a dimensionality reduction operator a PCA with the number of components that keeps the 95% of the variance.*

cycle) to its end ($\geq$ 2018, bottom-left, end of the 24th solar cycle), time follows a coherent horseshoe path. Figure 11.8b organizes the magnetograms by the four solar activity classes, highlighting the presence of clusters of magnetograms according to the activity phase. Future studies with more data from the next and subsequent solar cycles (from 25th on) would allow characterizing the transition dynamics in this feature space, providing new tools for analyzing solar phenomena.

Figure 11.9 analyses the classification capability of a Gaussian Support Vector Machine (SVM) supervised classifier trained on the features extracted by the selected architecture. In particular, Figure 11.9a shows the confusion matrix, highlighting that the few errors are of consecutive solar activity phases. More interestingly, Figure 11.9b plots such errors over time, revealing that such errors are almost all in the nearby of a transition boundary. This is a consequence of the sharp boundaries introduced during the classes definition (see Section 11.2.2), whereas the physical process has gradual transitions with local variations within the global trends.

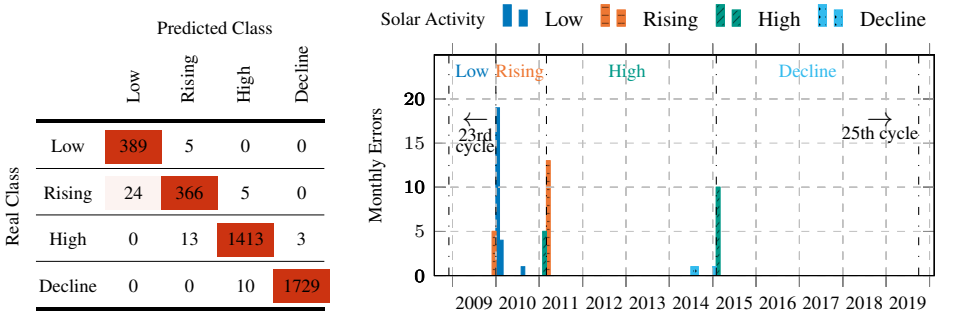
The final step of the analysis aims at exploiting the temporal structure highlighted in Figure 11.8, with magnetograms defining a horseshoe path over the years. To do so, two unsupervised clustering algorithms (k-Means and GMM) have been taken into account and trained with the goal of defining four clusters on a balanced subset of the magnetograms. Ideally, the generated clusters should match the solar activity classes defined in Section 11.2.2 and shown in Figures 11.5 and 11.8b. Figure 11.10 shows the results of this analysis by taking into account all the samples in the analysis (the samples not present in the "training" subset are assigned to the closest cluster). There is good correspondence between the original classes and the generated clusters for both the algorithms, with several magnetograms in the nearby of a transition phase assigned to the adjacent solar activity class, as expected after introducing sharp boundaries. Figure 11.10c shows the distribution of the samples in each cluster, revealing that only the class *Rising* is the only class that has



(a) *A subset of the samples in Figure 11.8b labelled by years.*

(b) *The samples labelled by the four solar activity classes.*

Figure 11.8: *The distribution of solar magnetograms on the first two dimensions (those having the highest variance) of the features space defined by the ResNet output of the layer conv4_0, then reduced by a PCA with 3468 components, i.e., those keeping 95% of the variance. Interestingly, both the figures shows a path in the feature space during the years (and thus the phases) of the solar cycle.*

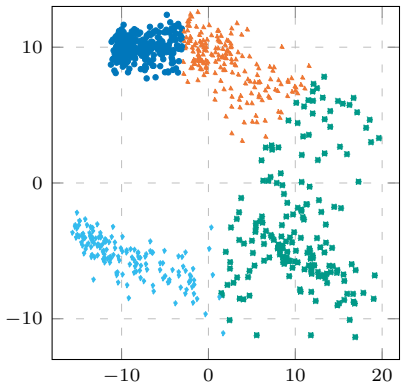


(a) *The confusion matrix of the Gaussian SVM.*

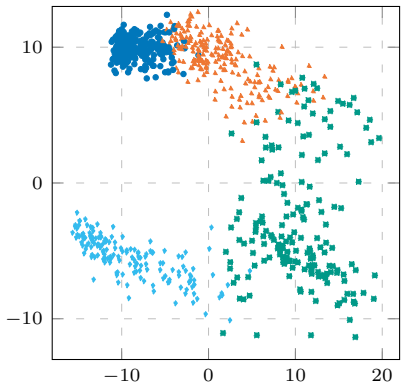
(b) *The distribution of errors across time. For each month the number of errors is plotted, if presents.*

Figure 11.9: *The solar activity classification results of a Gaussian SVM on the features space defined by the ResNet output of layer conv4_0, then reduced by a PCA with 3468 components, i.e., those keeping 95% of the variance. Interestingly, almost all the errors are near the transition phases.*

not been clearly isolated since the cluster containing the majority of its samples comprises mostly magnetograms belonging to the *High* class. This behavior, more severe with the k-Means algorithm, might be due to the unbalanced nature of the evaluation set but also



(a) *k-Means clustering results.*



(b) *GMM clustering results.*

		Real Class			
	Cluster	Low	Rising	High	Decline
k-Means	●	**395**	165	4	0
	▲	5	288	**420**	0
	■	0	0	**1011**	329
	◆	0	0	2	**1337**

		Real Class			
	Cluster	Low	Rising	High	Decline
GMM	●	**388**	115	2	0
	▲	12	338	**383**	0
	■	0	0	**1049**	318
	◆	0	0	3	**1348**

(c) *The distribution of the samples in the clusters.*

Figure 11.10: *The solar magnetograms clustering results on the first two dimensions (those having the highest variance) of the features space defined by the ResNet output of the layer conv4_0, then reduced by a PCA with 3468 components, i.e., those keeping* 95% *of the variance. For both the k-Means (Figure 11.10a) and the GMM (Figure 11.10b) algorithms, the number of samples is balanced on the four solar activity classes (randomly subsampled). Finally, Figure 11.10c compares the clusters defined by the two considered algorithms with the expected classes (Figure 11.8b).*

by a significant part of *Rising* magnetograms that are clustered into the *Low* one.

Part VI

Conclusions

CHAPTER 12

Conclusions and Future Work

Tiny Machine Learning is a relatively new research area aiming at designing machine and deep learning solutions that can be executed on embedded systems, IoT, or microcontroller units, i.e., with a memory footprint of a few kilobytes and energy consumption of milli- or micro-Watts. Moreover, the solutions available in the related literature primarily focus on supporting the inference of such Tiny Machine Learning algorithms on the device, whereas there are very few works addressing the problem of on-device learning.

In this scenario, this work came into play by proposing a methodology to design and deploy Deep and Wide Tiny Machine Learning solutions, where the term deep refers to the support for deep learning models, whereas the term wide to the possibility to distribute the deep learning pipeline across possibly heterogeneous embedded systems, IoT or microcontroller units. In the remaining of the section the main results, as well as possible future research directions, are summarized.

12.1 On-Device Deep Tiny Machine Learning

The first aspect the methodology takes into account is the possibility to design approximated Deep Learning Models by means of structured pruning, i.e., with task dropping, or precision scaling, i.e., by quantizing or relying on low-precision representations for the deep learning model's parameters. This approach leads to deep learning models with a memory footprint that satisfies the technological memory constraints introduced by the embedded system, IoT, or microcontroller unit the deep learning model will be executed

on. Moreover, the resulting deep learning model will have a reduced computational load as well.

In addition, the methodology encompassed as a possible design choice the introduction of Early-Exits, here via Gate-Classification, that can reduce the mean computational load by introducing a gating mechanism at one or more intermediate points within the deep learning pipeline. In such gates, if the processing pipeline has enough confidence about the final outcome can directly provide it without the need to execute the remaining layers.

Finally, with the goal of supporting on-device learning, a first work aiming at making deep learning models adaptive in presence of concept drift has been proposed. This active approach relies on a Change Detection Test on the deep learning model classification error and on an adaptation procedure that identifies the portion of the deep learning processing pipeline that has become obsolete after the concept drift. Hence, the adaptation does not require a complete re-training of the pipeline in response to a change. After that, the methodology included the first approach to on-device learning, i.e., a solution based on a fixed deep learning-based feature extractor and an adaptive k-Nearest Neighbors classifier. In particular, such an approach has been tailored to all the three approaches known in the literature of concept drift, i.e., passive, active, and hybrid.

Future research directions include the possibility to encompass unstructured pruning mechanisms as well as the (at least partial and constrained) exploration of feasible neural architectures, as in Neural Architecture Search approaches (Elsken et al., 2019; Li and Talwalkar, 2020). Moreover, there is room for improvement also in the on-device algorithm by exploring memory control mechanisms in the passive algorithm, unsupervised change detection tests in the active (and hybrid) one(s), as well as the introduction of precision scaling mechanisms or the conversion of the non-parametric adaptive classifier to a parametric one.

12.2 Deep Wide Tiny Machine Learning

The introduction of Wide Tiny Machine Learning solutions extended the plethora of the proposed methodology's possible applications. In particular, there is the possibility to split the deep learning pipeline into smaller and simpler tasks that are distributed into a heterogeneous network of embedded systems, IoT or microcontroller units. The methodology encompassed single deep learning models with or without early exits, multiple (and possibly different) deep learning models, and deep learning models that share part of their processing pipeline.

Moreover, an approach for encrypted computation on the Cloud has been proposed to deal with all those scenarios in which there is still the need to rely on the Cloud computation and, at the same time, the processed data are sensible and should not be in any way decrypted or sniffed by the Cloud.

Future research directions include the management of network failures in the methodology, a more accurate modeling of the data transmissions, the possibility to manage the amount of data transferred among IoT units, the possibility of the IoT units to move within, to leave from or to enter into the network, the definition of one or more algorithms computing the distribution of the deep learning pipeline (e.g., heuristic for fast but maybe sub-optimal mappings, branch and bound ones for a more accurate exploration of the solutions space with bounds on the exploration time), and the management of the energy

from other perspectives (for instance, the possibility for the IoT units to harvest energy over time).

12.3 From the Laboratory to the Wild

Finally, some of the techniques proposed in the methodology for designing Deep and Wide Tiny Machine Learning solutions have been applied to two real-world scenarios. At first, the detection of bird calls within audio waveforms in remote and wild environments. In this domain, the proposed solution with less than 512 KB of memory footprint and a processing time of a bit more than 3 s (for waveforms of 10 s) can operate for more than a week on a common micro-controller, the STM32H7, with a $2\,000$ mAh battery. The second scenario is the prediction of the solar activity from the acquired magnetograms. In this field, a characterization of the phenomena has been carried out, with significant insights on the problem as well as the definition of a feature space where all the four identified solar activity phases within a single solar cycle can be clearly identified.

Here, future research directions employ the possibility to deal with the recognition of the birds within the acquired waveforms, e.g., by a two-step classifier in which the first step is the proposed detection algorithm and the second step aims at identifying the bird species. In the scenario of sun activity prediction, the introduction of predictive algorithms starting from the proposed characterization might be beneficial in daily life if they can anticipate solar activity.

Bibliography

Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):79, 2018. (Cited on page 88)

Kamil Adamczewski and Mijung Park. Dirichlet pruning for convolutional neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 3637–3645. PMLR, 2021. (Cited on page 24)

Toni Adame, Albert Bel, Boris Bellalta, Jaume Barcelo, and Miquel Oliver. Ieee 802.11 ah: the wifi approach for m2m communications. *IEEE Wireless Communications*, 21(6):144–152, 2014. (Cited on page 118)

Charu C Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618. VLDB Endowment, 2006. (Cited on page 30)

Ankur Agrawal, Silvia M Mueller, Bruce M Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. Dlfloat: A 16-b floating point format designed for deep learning training and inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 92–95. IEEE, 2019. (Cited on page 26)

T Mitchell Aide, Carlos Corrada-Bravo, Marconi Campos-Cerqueira, Carlos Milan, Giovany Vega, and Rafael Alvarez. Real-time bioacoustics monitoring and automated species identification. *PeerJ*, 1:e103, 2013. (Cited on page 134)

Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015. (Cited on page 22)

Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016. (Cited on page 23)

Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394, 2017. (Cited on page 23)

Cesare Alippi. *Intelligence for Embedded Systems: A Methodological Approach*. Springer, 2014. (Cited on page 47)

Cesare Alippi, Giacomo Boracchi, and Manuel Roveri. Just-in-time classifiers for recurrent concepts. *IEEE transactions on neural networks and learning systems*, 24(4):620–634, 2013. (Cited on pages 30 and 61)

Cesare Alippi, Giacomo Boracchi, and Manuel Roveri. Hierarchical change-detection tests. *IEEE Transactions on Neural Networks and Learning Systems*, 28(2):246–258, 2017. (Cited on page 54)

Cesare Alippi, Simone Disabato, and Manuel Roveri. Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case. In *17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 212–223, Porto, apr 2018. IEEE. (Cited on pages 4, 7, 13, 14, 22, 24, 31, 60, 94, and 99)

Paulo RL Almeida, Luiz S Oliveira, Alceu S Britto Jr, and Robert Sabourin. Adapting dynamic classifier selection for concept drift. *Expert Systems with Applications*, 104:67–85, 2018. (Cited on page 30)

Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. (Cited on pages 36 and 61)

Jose M Alvarez and Mathieu Salzmann. Compression-aware training of deep networks. *Advances in neural information processing systems*, 30:856–867, 2017. (Cited on page 24)

Sven E Anderson, Amish S Dave, and Daniel Margoliash. Template-based automatic recognition of birdsong syllables from continuous recordings. *The Journal of the Acoustical Society of America*, 100(2):1209–1219, 1996. (Cited on page 134)

Kota Ando, Kodai Ueyoshi, Kentaro Orimo, Haruyoshi Yonekawa, Shimpei Sato, Hiroki Nakahara, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, Tadahiro Kuroda, et al. Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w. *IEEE Journal of Solid-State Circuits*, 53(4):983–994, 2017. (Cited on pages 22 and 23)

Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241. IEEE, 2016. (Cited on page 23)

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1131–1135. IEEE, 2015. (Cited on pages 26 and 27)

Parinita Badre, S Bandiwadekar, P Chandanshive, Aakanksha Chaudhari, and Mrs Sonali Jadhav. Automatically identifying animals using deep learning. *Int. J. Recent Innov. Trends Comput. Commun*, 6(4):194–197, 2018. (Cited on page 4)

Manuel Baena-Garcıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavalda, and Rafael Morales-Bueno. Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, volume 6, pages 77–86, 2006. (Cited on pages 30 and 67)

Alexei Baevski, Steffen Schneider, and Michael Auli. vq-wav2vec: Self-supervised learning of discrete speech representations. In *International Conference on Learning Representations*, 2019. (Cited on page 3)

Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020. (Cited on page 28)

Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3, 2021. (Cited on pages 31 and 135)

Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020. (Cited on pages 4, 5, and 31)

Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 5151–5159, 2018. (Cited on page 26)

Mauro Barni, Claudio Orlandi, and Alessandro Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151. ACM, 2006. (Cited on page 90)

Michèle Basseville, Igor V Nikiforov, et al. *Detection of abrupt changes: theory and application*, volume 104. Prentice Hall Englewood Cliffs, 1993. (Cited on pages 53 and 54)

Selin Bastas, Mohammad Wadood Majid, Golrokh Mirzaei, Jeremy Ross, Mohsin M Jamali, Peter V Gorsevski, Joseph Frizado, and Verner P Bingman. A novel feature extraction algorithm for classification of bird flight calls. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1676–1679. IEEE, 2012. (Cited on page 134)

Franz Berger, William Freillinger, Paul Primus, and Wolfgang Reisinger. Bird audio detection - dcase 2018. Technical report, DCASE2018 Challenge, September 2018. (Cited on pages 135, 142, and 143)

Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*, 2019. (Cited on page 28)

Kartikeya Bhardwaj, Ching-Yi Lin, Anderson Sartor, and Radu Marculescu. Memory-and communication-aware model compression for distributed deep learning inference on iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019. (Cited on pages 5 and 33)

Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–23, 2018. (Cited on pages 22 and 23)

Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 45–56, 2019a. (Cited on page 90)

Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 3–13. ACM, 2019b. (Cited on page 88)

Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 527–536, 2017. (Cited on pages 5, 24, 28, 29, 74, and 119)

Andrea Bona, Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, Vittorio Zaccaria, and Roberto Zafalon. Reducing the complexity of instruction-level power models for vliw processors. *Design automation for embedded systems*, 10(1):49–67, 2005. (Cited on page 19)

Carlo Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, 8:3–62, 1936. (Cited on page 56)

Giacomo Boracchi, Diego Carrera, Cristiano Cervellera, and Danilo Maccio. Quanttree: Histograms for change detection in multivariate data streams. In *International Conference on Machine Learning*, pages 639–648. PMLR, 2018. (Cited on page 30)

Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018. (Cited on page 90)

Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. Embracing diversity: Enhanced dsp blocks for low-precision deep learning on fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 35–357. IEEE, 2018. (Cited on page 22)

Arash Bozorgchenani, Simone Disabato, Daniele Tarchi, and Manuel Roveri. An energy harvesting solution for computation offloading in fog computing networks. *Computer Communications*, 160: 577–587, 2020. (Cited on pages 32, 33, and 84)

Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014. (Cited on pages 89 and 90)

Forrest Briggs, Raviv Raich, and Xiaoli Z Fern. Audio classification of bird species: A statistical manifold approach. In *2009 Ninth IEEE International Conference on Data Mining*, pages 51–60. IEEE, 2009. (Cited on page 134)

Stuart A Brooker, Philip A Stephens, Mark J Whittingham, and Stephen G Willis. Automated detection and classification of birdsong: An ensemble approach. *Ecological Indicators*, 117: 106609, 2020. (Cited on page 135)

Li Bu, Cesare Alippi, and Dongbin Zhao. A pdf-free change detection test based on density difference estimation. *IEEE transactions on neural networks and learning systems*, 29(2):324–334, 2016. (Cited on page 30)

Johannes Buchner. Synthetic speech commands: A public dataset for single-word speech recognition. *Dataset available from https://www.kaggle.com/jbuchner/synthetic-speech-commands-dataset/*, 2017. (Cited on page 108)

Adrian Bulat and Georgios Tzimiropoulos. Bit-mixer: Mixed-precision networks with runtime bit-width selection. *arXiv preprint arXiv:2103.17267*, 2021. (Cited on pages 5 and 27)

Grégoire Burel, Hassan Saif, and Harith Alani. Semantic wide and deep learning for detecting crisis-information categories on social media. In *International semantic web conference*, pages 138–155. Springer, 2017. (Cited on page 6)

Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91. IEEE, 2019. (Cited on page 26)

C Sidney Burrus and TW Parks. *Convolution Algorithms*. Citeseer, 1985. (Cited on page 15)

Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems*, 33, 2020a. (Cited on pages 5, 31, 32, and 59)

Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020b. (Cited on page 27)

Zhaowei Cai and Nuno Vasconcelos. Rethinking differentiable search for mixed-precision neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. (Cited on page 27)

Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5918–5926, 2017. (Cited on pages 5 and 27)

Leopold Cambier, Anahita Bhiwandiwalla, Ting Gong, Oguz H. Elibol, Mehran Nekuii, and Hanlin Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. In *International Conference on Learning Representations*, 2020. (Cited on page 26)

Alberto Cano. A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(1):e1232, 2018. (Cited on page 22)

Miguel A Carreira-Perpinán and Yerlan Idelbayev. "learning-compression" algorithms for neural net pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8532–8541, 2018. (Cited on page 24)

Juan Castillo, Héctor Posadas, Eugenio Villar, and Marcos Martínez. Energy consumption estimation technique in embedded processors with stable power consumption based on source-code operator energy figures. In *XXII conference on design of circuits and integrated systems*, page 1, 2007. (Cited on pages 18 and 19)

Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, 2016. (Cited on page 4)

Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*, pages 2722–2730, 2015. (Cited on page 3)

Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018a. (Cited on page 4)

Jianguo Chen, Kenli Li, Qingying Deng, Keqin Li, and S Yu Philip. Distributed deep learning model for intelligent video surveillance systems with edge computing. *IEEE Transactions on Industrial Informatics*, 2019a. (Cited on pages 5 and 33)

Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15834–15846. Curran Associates, Inc., 2020. (Cited on page 26)

Wei-Hao Chen, Kai-Xiang Li, Wei-Yu Lin, Kuo-Hsiang Hsu, Pin-Yi Li, Cheng-Han Yang, Cheng-Xin Xue, En-Yu Yang, Yen-Kai Chen, Yun-Sheng Chang, et al. A 65nm 1mb nonvolatile computing-in-memory reram macro with sub-16ns multiply-and-accumulate for binary dnn ai edge processors. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 494–496. IEEE, 2018b. (Cited on page 23)

Weihan Chen, Peisong Wang, and Jian Cheng. Towards mixed-precision quantization of neural networks via constrained optimization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5350–5359, October 2021. (Cited on pages 27 and 28)

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014. (Cited on page 23)

Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Shuicheng Yan, and Jiashi Feng. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3435–3444, 2019b. (Cited on page 24)

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016. (Cited on page 6)

Zixue Cheng, Peng Li, Junbo Wang, and Song Guo. Just-in-time code offloading for wearable computing. *IEEE Transactions on Emerging Topics in Computing*, 3(1):74–83, 2015. (Cited on page 33)

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017. (Cited on pages 89 and 90)

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. (Cited on pages 15 and 141)

Po-Han Chi, Pei-Hung Chung, Tsung-Han Wu, Chun-Cheng Hsieh, Yen-Hao Chen, Shang-Wen Li, and Hung-yi Lee. Audio albert: A lite bert for self-supervised learning of audio representation. In *2021 IEEE Spoken Language Technology Workshop (SLT)*, pages 344–350. IEEE, 2021. (Cited on page 26)

François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017. (Cited on pages 24 and 25)

Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro*, 38(2):8–20, 2018. (Cited on page 22)

Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. Xnor neural engine: A hardware accelerator ip for 21.6-fj/op binary neural network inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2940–2951, 2018. (Cited on page 23)

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. (Cited on pages 36 and 148)

European Parliament Council of European Union. Regulation (eu) no 2016/679, article 4(1), 2016. (Cited on page 88)

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014. (Cited on page 26)

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015. (Cited on pages 23 and 27)

Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010. (Cited on page 33)

Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016. (Cited on page 22)

Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018. (Cited on page 26)

Tamraparni Dasu, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *In Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*, 2006. (Cited on page 30)

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012. (Cited on page 105)

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. (Cited on pages 103, 108, and 136)

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014. (Cited on pages 5 and 25)

Tim Dettmers. 8-bit approximations for parallelism in deep learning. In *4th International Conference on Learning Representations, ICLR 2016*, San Juan, Puerto Rico, May 2–4 2016. (Cited on page 26)

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019. (Cited on pages 3 and 28)

Simone Disabato and Manuel Roveri. Reducing the computation load of convolutional neural networks through gate classification. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018. (Cited on pages 5, 7, 24, 28, 74, 102, and 119)

Simone Disabato and Manuel Roveri. Learning convolutional neural networks in presence of concept drift. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019. (Cited on pages 7, 30, 52, 67, and 105)

Simone Disabato and Manuel Roveri. Incremental on-device tiny machine learning. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, pages 7–13, 2020. (Cited on pages 5, 8, 31, 32, 59, 60, 107, and 135)

Simone Disabato and Manuel Roveri. Tiny machine learning for concept drift. *arXiv preprint arXiv:2107.14759*, 2021. (Cited on pages 5, 8, 31, 32, 59, and 107)

Simone Disabato, Alessandro Falcetta, Alessio Mongelluzzo, and Manuel Roveri. A privacy-preserving distributed architecture for deep-learning-as-a-service. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020. (Cited on pages 9, 88, 91, 126, and 129)

Simone Disabato, Giuseppe Canonaco, Paul G Flikkema, Manuel Roveri, and Cesare Alippi. Birdsong detection at the edge with deep learning. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 9–16. IEEE, 2021a. (Cited on pages 10 and 113)

Simone Disabato, Manuel Roveri, and Cesare Alippi. Distributed deep convolutional neural networks for the internet-of-things. *IEEE Transactions on Computers*, 2021b. (Cited on pages 4, 5, 9, 28, 33, 73, 74, and 117)

Gregory Ditzler and Robi Polikar. Hellinger distance based drift detection for nonstationary environments. In *Computational Intelligence in Dynamic and Uncertain Environments (CIDUE), 2011 IEEE Symposium on*, pages 41–48. IEEE, 2011. (Cited on page 30)

Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, 2015. (Cited on pages 5, 29, 35, 52, and 59)

Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4860–4874, 2017. (Cited on page 24)

Piotr Duda, Leszek Rutkowski, Maciej Jaworski, and Danuta Rutkowska. On the parzen kernel-based probability density function learning procedures over time-varying streaming data with applications to pattern classification. *IEEE transactions on cybernetics*, 50(4):1683–1696, 2018. (Cited on page 30)

Olivier Dufour, Thierry Artieres, Hervé Glotin, and Pascale Giraudet. Clusterized mel filter cepstral coefficients and support vector machines for bird song identification. *Proceedings of the 1st workshop on Machine Learning for Bioacoustics joint to the 30th ICML*, pages 89–93, 2013. (Cited on page 134)

Aysegul Dundar, Jonghoon Jin, Berin Martini, and Eugenio Culurciello. Embedded streaming deep neural networks accelerator with applications. *IEEE transactions on neural networks and learning systems*, 28(7):1572–1583, 2016. (Cited on pages 4, 23, and 31)

Olive Jean Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961. (Cited on page 56)

Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985. (Cited on page 90)

Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. Deepshift: Towards multiplication-less neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2359–2368, 2021. (Cited on page 26)

Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14629–14638, 2020. (Cited on page 25)

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019. (Cited on pages 50 and 156)

Jeremy Elson, John JD Douceur, Jon Howell, and Jared Saul. Asirra: a captcha that exploits interest-aligned manual image categorization. In *Proceedings of 14th ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, Inc., October 2007. (Cited on pages 103, 104, and 105)

Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22(10):1517–1531, 2011. (Cited on page 29)

Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology & architecture*. Pearson Education, 2013. (Cited on page 87)

Fabio Fabris and Alex A Freitas. Analysing the overfit of the auto-sklearn automated machine learning tool. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 508–520. Springer, 2019. (Cited on page 49)

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *8th International Conference on Learning Representations, ICLR 2020*, Addis Ababa, Ethiopia, April 26-30 2020. (Cited on page 26)

Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012. (Cited on pages 88 and 90)

Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Cvpr 2011 Workshops*, pages 109–116. IEEE, 2011. (Cited on page 23)

Laith Farhan, Sinan T Shukur, Ali E Alissa, Mohmad Alrweg, Umar Raza, and Rupak Kharel. A survey on the challenges and opportunities of the internet of things (iot). In *2017 Eleventh International Conference on Sensing Technology (ICST)*, pages 1–5. IEEE, 2017. (Cited on page 83)

Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems*, 32, 2019. (Cited on pages 5 and 31)

Igor Fedorov, Marko Stamenovic, Carl Jensen, Li-Chia Yang, Ari Mandell, Yiming Gan, Matthew Mattina, and Paul N Whatmough. Tinylstms: Efficient neural speech enhancement for hearing aids. *arXiv preprint arXiv:2005.11138*, 2020. (Cited on pages 5 and 31)

Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. A hybrid energy-estimation technique for extensible processors. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 23(5):652–664, 2004. (Cited on page 18)

André C Ferreira, Liliana R Silva, Francesco Renna, Hanja B Brandl, Julien P Renoult, Damien R Farine, Rita Covas, and Claire Doutrelant. Deep learning-based methods for individual recognition in small birds. *Methods in Ecology and Evolution*, 11(9):1072–1085, 2020. (Cited on page 135)

Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018. (Cited on page 22)

Elizabeth JS Fox, J Dale Roberts, and Mohammed Bennamoun. Text-independent speaker identification in birds. In *Ninth International Conference on Spoken Language Processing*, 2006. (Cited on page 134)

Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999. (Cited on page 30)

Karl-Heinz Frommolt and Klaus-Henry Tauchert. Applying bioacoustic methods for long-term monitoring of a nocturnal wetland bird. *Ecological Informatics*, 21:4–12, 2014. (Cited on page 134)

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. *Advances in neural information processing systems*, 29:1019–1027, 2016. (Cited on page 25)

João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46(4):44, 2014. (Cited on page 29)

Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. Learning to fly by crashing. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3948–3955. IEEE, 2017. (Cited on page 3)

Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. Compressing large-scale transformer-based models: A case study on bert. *Transactions of the Association for Computational Linguistics*, 9: 1061–1080, 2021. (Cited on page 26)

Chang Gao, Stefan Braun, Ilya Kiselev, Jithendar Anumula, Tobi Delbruck, and Shih-Chii Liu. Real-time speech recognition for iot purpose using a delta recurrent neural network accelerator. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019. (Cited on page 22)

Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1536414.1536440. (Cited on page 90)

Pradipta Ghosh, Quynh Nguyen, and Bhaskar Krishnamachari. Container orchestration for dispersed computing. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, pages 19–24, 2019. (Cited on page 33)

Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016. (Cited on pages 90 and 93)

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010. (Cited on page 56)

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. (Cited on pages 27 and 28)

Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. Compiling kb-sized machine learning models to tiny iot devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–95, 2019. (Cited on pages 5 and 31)

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1586–1595, 2018. (Cited on page 24)

Mitchell Gordon, Kevin Duh, and Nicholas Andrews. Compressing bert: Studying the effects of weight pruning on transfer learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*, pages 143–155, 2020. (Cited on page 26)

Martin Graciarena, Michelle Delplanche, Elizabeth Shriberg, Andreas Stolcke, and Luciana Ferrer. Acoustic front-end optimization for bird species recognition. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 293–296. IEEE, 2010. (Cited on page 134)

Benjamin Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014. (Cited on page 16)

Thomas Grill and Jan Schlüter. Two convolutional neural networks for bird detection in audio signals. In *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 1764–1768. IEEE, 2017. (Cited on page 135)

Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019. (Cited on page 22)

Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE transactions on computer-aided design of integrated circuits and systems*, 37(1):35–47, 2017a. (Cited on pages 22 and 23)

Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of fpga-based neural network accelerator. *arXiv preprint arXiv:1712.08934*, 2017b. (Cited on page 22)

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015. (Cited on pages 5, 13, and 26)

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR 2016*, San Juan, Puerto Rico, May 2-4 2016. (Cited on pages 5, 24, and 26)

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017. (Cited on page 22)

Patrick J Hart, Esther Sebastián-González, Ann Tanimoto, Alia Thompson, Tawn Speetjens, Madolyn Hopkins, and Michael Atencio-Picado. Birdsong characteristics are related to fragment size in a neotropical forest. *Animal Behaviour*, 137:45–52, 2018. (Cited on page 138)

Peter Hart. The condensed nearest neighbor rule (corresp.). *IEEE transactions on information theory*, 14(3):515–516, 1968. (Cited on pages XV, 62, and 63)

Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5353–5360, 2015. (Cited on pages 13 and 14)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. (Cited on pages 3, 25, 51, 108, 135, 136, and 149)

Wenchen He, Shaoyong Guo, Song Guo, Xuesong Qiu, and Feng Qi. Joint dnn partition deployment and resource allocation for delay-sensitive deep learning inference in iot. *IEEE Internet of Things Journal*, 7(10):9241–9254, 2020. (Cited on page 33)

Y He, G Kang, X Dong, Y Fu, and Y Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *IJCAI International Joint Conference on Artificial Intelligence*, 2018. (Cited on page 24)

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017. (Cited on pages 5 and 24)

Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 69–76. IEEE, 2019. (Cited on page 26)

Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 133–136. IEEE, 2020. (Cited on page 22)

Frank Hill. The global oscillation network group facilit–an example of research to operations in space weather. *Space Weather*, 16(10):1488–1497, 2018. (Cited on pages 145 and 146)

Ivan Himawan, Michael Towsey, and Paul Roe. 3d convolution recurrent neural networks for bird sound detection. In *Proceedings of the 3rd Workshop on Detection and Classification of Acoustic Scenes and Events*, pages 1–4, 2018. (Cited on page 135)

Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 15–20, New York, NY, USA, 2013. ACM. (Cited on page 33)

Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014. (Cited on pages 18 and 19)

M Shamim Hossain and Ghulam Muhammad. Cloud-assisted speech and face recognition framework for health monitoring. *Mobile Networks and Applications*, 20(3):391–399, 2015. (Cited on page 87)

Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33, 2020. (Cited on page 26)

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. (Cited on pages 24 and 25)

Diyi Hu and Bhaskar Krishnamachari. Throughput optimized scheduler for dispersed computing systems. In *2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 76–84. IEEE, 2019. (Cited on page 33)

Diyi Hu and Bhaskar Krishnamachari. Fast and accurate streaming cnn inference via communication compression on the edge. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 157–163. IEEE, 2020. (Cited on pages 5 and 33)

Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 304–320, 2018. (Cited on pages 24 and 25)

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017. (Cited on pages 23 and 27)

Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. ACM. (Cited on page 29)

Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014. (Cited on page 27)

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. (Cited on pages 24 and 25)

Allison S Injaian, Lauren Y Poon, and Gail L Patricelli. Effects of experimental anthropogenic noise on avian settlement patterns and reproductive success. *Behavioral Ecology*, 29(5):1181–1189, 2018. (Cited on page 133)

Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. In *4th International Conference on Learning Representations, ICLR 2016*, San Juan, Puerto Rico, May 2-4 2015. (Cited on page 25)

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015. (Cited on page 16)

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. (Cited on page 26)

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014. (Cited on pages 24 and 25)

Qing Jin, Linjie Yang, and Zhenyu Liao. Adabits: Neural network quantization with adaptive bit-widths. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2146–2156, 2020. (Cited on page 27)

Jeff Johnson. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018. (Cited on page 26)

Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, 2018. (Cited on page 22)

Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018. (Cited on page 90)

Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017. (Cited on page 33)

Yi-Hsuan Kao, Bhaskar Krishnamachari, Moo-Ryong Ra, and Fan Bai. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, 16(11):3056–3069, 2017. (Cited on page 32)

Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972. (Cited on page 78)

Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*, pages 3301–3310. PMLR, 2019. (Cited on pages 5 and 28)

Ashish Khetan and Zohar Karnin. schubert: Optimizing elements of bert. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2807–2818, 2020. (Cited on page 26)

Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. Novel dataset for fine-grained image categorization: Stanford dogs. *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, 2(1), 2011. (Cited on pages 103 and 104)

Hyeonuk Kim, Jaehyeong Sim, Yeongjae Choi, and Lee-Sup Kim. Nand-net: Minimizing computational complexity of in-memory processing for binary neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 661–673. IEEE, 2019. (Cited on page 23)

Minje Kim and Paris Smaragdis. Bitwise neural networks for efficient single-channel source separation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 701–705. IEEE, 2018. (Cited on page 27)

Mirae Kim, Jungkeol Lee, Youngil Kim, and Yong Ho Song. An analysis of energy consumption under various memory mappings for fram-based iot devices. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 574–579. IEEE, 2018. (Cited on page 4)

Suchang Kim, Jihyuck Jo, and In-Cheol Park. Hybrid convolution architecture for energy-efficient deep neural network processing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(5):2017–2029, 2021. (Cited on page 23)

Ralf Klinkenberg. Learning drifting concepts: Example selection vs. example weighting. *Intelligent data analysis*, 8(3):281–300, 2004. (Cited on page 30)

Phil C Knag, Gregory K Chen, H Ekin Sumbul, Raghavan Kumar, Steven K Hsu, Amit Agarwal, Monodeep Kar, Seongjong Kim, Mark A Anders, Himanshu Kaul, et al. A 617-tops/w all-digital binary neural network accelerator in 10-nm finfet cmos. *IEEE Journal of Solid-State Circuits*, 56 (4):1082–1092, 2020. (Cited on pages 22 and 23)

Joseph A Kogan and Daniel Margoliash. Automated recognition of bird song elements from continuous recordings using dynamic time warping and hidden markov models: A comparative study. *The Journal of the Acoustical Society of America*, 103(4):2185–2196, 1998. (Cited on page 134)

Chih-Yuan Koh, Jaw-Yuan Chang, Chiang-Lin Tai, Da-Yo Huang, Han-Hsing Hsieh, and Yi-Wen Liu. Bird sound classification using convolutional neural networks. In *CLEF (Working Notes)*, 2019. (Cited on page 135)

Urs Köster, Tristan J Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William H Constable, Oğuz H Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: an adaptive numerical format for efficient training of deep neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1740–1750, 2017. (Cited on page 26)

Bartosz Krawczyk and Michał Woźniak. One-class classifiers with incremental learning and forgetting for data streams with concept drift. *Soft Computing*, 19(12):3387–3400, 2015. (Cited on page 29)

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images, 2009. (Cited on pages 103 and 104)

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, volume 1 of *NIPS '12*, pages 1097–1105. Curran Associates Inc., 2012. (Cited on pages 16, 25, 27, 56, 99, 105, and 149)

Aayan Kumar, Vivek Seshadri, and Rahul Sharma. Shiftry: Rnn inference in 2kb of ram. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020. (Cited on page 31)

Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*, pages 1935–1944, 2017. (Cited on pages 5 and 31)

Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. Fastgrnn: a fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 9031–9042, 2018. (Cited on page 31)

Kim Laine. Simple encrypted arithmetic library 2.3.1. Technical report, Microsoft Research, WA, USA, 2017. (Cited on pages 89 and 94)

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2019. (Cited on page 29)

Hamed F Langroudi, Vedant Karia, Zachariah Carmichael, Abdullah Zyarah, Tej Pandit, John L Gustafson, and Dhireesha Kudithipudi. Alps: Adaptive quantization of deep neural networks with generalized posits. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3100–3109, 2021. (Cited on page 27)

Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020. (Cited on page 33)

Mario Lasseck. Bird song classification in field recordings: winning solution for nips4b 2013 competition. In *Proc. of int. symp. Neural Information Scaled for Bioacoustics, sabiod. org/nips4b, joint to NIPS, Nevada*, pages 176–181, 2013. (Cited on page 134)

Mario Lasseck. Acoustic bird detection with deep convolutional neural networks. In *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2018 Workshop (DCASE2018)*, pages 143–147, 2018. (Cited on pages 135, 142, and 143)

Johann Laurent, Eric Senn, Nathalie Julien, and Eric Martin. High-level energy estimation for dsp systems. In *PATMOS*, pages pp–3. IEEE, 2001. (Cited on page 19)

Jorma Laurikkala. Improving identification of difficult small classes by balancing class distribution. In *Conference on Artificial Intelligence in Medicine in Europe*, pages 63–66. Springer, 2001. (Cited on page 62)

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *3rd International Conference on Learning Representations, ICLR 2015*, San Diego, CA, USA, May 7-9 2014. (Cited on page 25)

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998. (Cited on pages 127 and 128)

Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. (Cited on page 148)

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. (Cited on pages 22, 103, and 104)

Chang-Hsing Lee, Sheng-Bin Hsu, Jau-Ling Shih, and Chih-Hsun Chou. Continuous birdsong recognition using gaussian mixture modeling of image shape features. *IEEE Transactions on Multimedia*, 15(2):454–464, 2012. (Cited on page 135)

Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 218–220. IEEE, 2018. (Cited on page 23)

Erich L Lehmann and Joseph P Romano. *Testing statistical hypotheses*. Springer Science & Business Media, 2006. (Cited on page 55)

Dawei Li, Xiaolong Wang, and Deguang Kong. Deeprebirth: Accelerating deep neural network execution on mobile devices. In *Thirty-second AAAI conference on artificial intelligence*, 2018. (Cited on page 5)

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR 2017*, Toulon, France, April 24-26 2017. (Cited on page 24)

HongLin Li, Payam Barnaghi, Shirin Enshaeifar, and Frieder Ganz. Continual learning using bayesian neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020a. (Cited on page 30)

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in artificial intelligence*, pages 367–377. PMLR, 2020. (Cited on pages 50 and 156)

Tuanhui Li, Baoyuan Wu, Yujiu Yang, Yanbo Fan, Yong Zhang, and Wei Liu. Compressing convolutional neural networks via factorized convolutional filters. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3977–3986, 2019. (Cited on pages 24 and 25)

Yawei Li, Shuhang Gu, Christoph Mayer, Luc Van Gool, and Radu Timofte. Group sparsity: The hinge between filter pruning and decomposition for network compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8018–8027, 2020b. (Cited on page 25)

Yuxi Li. Deep reinforcement learning. *arXiv preprint arXiv:1810.06339*, 2018. (Cited on page 3)

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. In *Proceedings of the International Conference on Machine Learning, ICML '20*, pages 5958–5968, 2020c. (Cited on page 26)

Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 70–79, 2021. (Cited on page 31)

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016. (Cited on pages 26, 27, and 28)

Ji Lin, Wei-Ming Chen, Yujun Lin, john cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11711–11722. Curran Associates, Inc., 2020a. (Cited on pages 4, 5, and 31)

Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1529–1538, 2020b. (Cited on page 24)

Shaohui Lin, Rongrong Ji, Chao Chen, Dacheng Tao, and Jiebo Luo. Holistic cnn compression via low-rank decomposition with knowledge transfer. *IEEE transactions on pattern analysis and machine intelligence*, 41(12):2889–2905, 2018. (Cited on page 5)

Ye Lin, Yanyang Li, Tengbo Liu, Tong Xiao, Tongran Liu, and Jingbo Zhu. Towards fully 8-bit integer inference for the transformer model. *arXiv preprint arXiv:2009.08034*, 2020c. (Cited on page 28)

Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015. (Cited on pages 24 and 25)

Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018. (Cited on page 27)

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017. (Cited on page 24)

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *7th International Conference on Learning Representations, ICLR 2019*, New Orleans, LA, USA, May 6-9 2019. (Cited on page 24)

W Livingston, MJ Penn, and L Svalgaard. Decreasing sunspot magnetic fields explain unique 10.7 cm radio flux. *The Astrophysical Journal Letters*, 757(1):L8, 2012. (Cited on page 148)

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2): 129–137, 1982. (Cited on page 148)

Yun Long, Taesik Na, and Saibal Mukhopadhyay. Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(12):2781–2794, 2018. (Cited on page 23)

Gary Lorden et al. Procedures for reacting to a change in distribution. *The Annals of Mathematical Statistics*, 42(6):1897–1908, 1971. (Cited on pages 30, 53, 54, and 66)

Viktor Losing, Barbara Hammer, and Heiko Wersing. Knn classifier with self adjusting memory for heterogeneous concept drift. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 291–300. IEEE, 2016. (Cited on page 61)

Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018. (Cited on page 29)

Yao Lu, Guangming Lu, Bob Zhang, Yuanrong Xu, and Jinxing Li. Super sparse convolutional neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4440–4447, 2019. (Cited on pages 24 and 25)

Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017. (Cited on page 24)

Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010. (Cited on page 89)

Yihuan Mao, Yujing Wang, Chufan Wu, Chen Zhang, Yang Wang, Quanlu Zhang, Yaming Yang, Yunhai Tong, and Jing Bai. Ladabert: Lightweight adaptation of bert through hybrid model compression. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 3225–3234, 2020. (Cited on page 26)

Peter Marler and Donald Isaac. Physical analysis of a simple bird song as exemplified by the chipping sparrow. *The Condor*, 62(2):124–135, 1960. (Cited on page 138)

Borja Martinez, Marius Monton, Ignasi Vilajosana, and Joan Daniel Prades. The power of models: Modeling power consumption for iot devices. *IEEE Sensors Journal*, 15(10):5777–5789, 2015. (Cited on page 4)

Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. Mixed low-precision deep learning inference using dynamic fixed point. *arXiv preprint arXiv:1701.08978*, 2017. (Cited on page 26)

Alberto Maria Metelli, Amarildo Likmeta, and Marcello Restelli. Propagating uncertainty in reinforcement learning via wasserstein barycenters. In *33rd Conference on Neural Information Processing Systems, NeurIPS 2019*, pages 4335–4347. Curran Associates, Inc., 2019. (Cited on page 3)

Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016. (Cited on pages 23, 46, and 47)

Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016. (Cited on pages 26 and 31)

Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017. (Cited on page 90)

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR, 2017. (Cited on page 25)

Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller. Characterizing the energy consumption of data transfers and arithmetic operations on x86- 64 processors. In *International conference on green computing*, pages 123–133. IEEE, 2010. (Cited on pages 18 and 19)

Bert Moons and Marian Verhelst. A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, 2016. (Cited on page 23)

Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247. IEEE, 2017. (Cited on page 22)

Bert Moons, Daniel Bankman, Lita Yang, Boris Murmann, and Marian Verhelst. Binareye: An always-on energy-accuracy-scalable binary cnn processor with all memory on chip in 28nm cmos. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE, 2018. (Cited on page 23)

Duncan JM Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 107–116, 2018. (Cited on page 23)

Rajdeep Mukherjee, Dipyaman Banerjee, Kuntal Dey, and Niloy Ganguly. Convolutional recurrent neural network based bird audio detection. Technical report, DCASE2018 Challenge, September 2018. (Cited on pages 135, 142, and 143)

Lorenz Muller, Julien Martel, and Giacomo Indiveri. Kernelized synaptic weight matrices. In *International Conference on Machine Learning*, pages 3654–3663. PMLR, 2018. (Cited on page 3)

Rafael Hernández Murcia and Vıctor Suárez Paniagua. Bird identification from continuous audio recordings. *Proceedings of the 1st workshop on Machine Learning for Bioacoustics joint to the 30th ICML*, pages 96–97, 2013. (Cited on page 134)

David Naccache and Jacques Stern. A new public key cryptosystem based on higher residues. In *ACM Conference on Computer and Communications Security*, pages 59–66. Citeseer, 1998. (Cited on page 90)

Lawrence Neal, Forrest Briggs, Raviv Raich, and Xiaoli Z Fern. Time-frequency segmentation of bird song in noisy acoustic environments. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2012–2015. IEEE, 2011. (Cited on page 134)

L Nikolić. On solutions of the pfss model with gong synoptic maps for 2006–2018. *Space Weather*, 17(8):1293–1311, 2019. (Cited on page 145)

Ljubomir Nikolić. Modelling the magnetic field of the solar corona with potential-field source-surface and schatten current sheet models. *Geological Survey of Canada, Open File*, 8007, 2017. (Cited on pages 145 and 146)

Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *International conference on the theory and applications of cryptographic techniques*, pages 308–318. Springer, 1998. (Cited on page 90)

Ewan S Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954. (Cited on pages 30, 53, and 66)

Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999. (Cited on page 90)

Kuldip K Paliwal, James G Lyons, and Kamil K Wójcicki. Preference for 20-40 ms window duration in speech analysis. In *2010 4th International Conference on Signal Processing and Communication Systems*, pages 1–4. IEEE, 2010. (Cited on page 138)

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019. (Cited on page 30)

Nikolaos Passalis, Jenni Raitoharju, Anastasios Tefas, and Moncef Gabbouj. Efficient adaptive inference for deep convolutional neural networks using hierarchical early exits. *Pattern Recognition*, 105:107346, 2020. (Cited on page 28)

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019. (Cited on page 149)

Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. (Cited on page 138)

Beatriz Pérez-Sánchez, Oscar Fontenla-Romero, and Bertha Guijarro-Berdiñas. A review of adaptive online learning for artificial neural networks. *Artificial Intelligence Review*, 49(2):281–299, 2018. (Cited on page 30)

Gordon JD Petrie. Solar magnetism in the polar regions. *Living Reviews in Solar Physics*, 12(1):5, 2015. (Cited on page 145)

Lena Pietruczuk, Leszek Rutkowski, Maciej Jaworski, and Piotr Duda. How to adjust an ensemble size in stream data mining? *Information Sciences*, 381:46–54, 2017. (Cited on page 30)

Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 31(4):497–508, 2001. (Cited on page 29)

Moshe Pollak. Average run lengths of an optimal method of detecting a change in distribution. *The Annals of Statistics*, pages 749–779, 1987. (Cited on page 54)

Jens Pomoell and Stefaan Poedts. Euhforia: European heliospheric forecasting information asset. *Journal of Space Weather and Space Climate*, 8:A35, 2018. (Cited on page 145)

Ilyas Potamitis. Automatic classification of a taxon-rich community recorded in the wild. *PloS one*, 9(5), 2014. (Cited on page 134)

Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 698–699, 2020. (Cited on page 27)

Nirosha Priyadarshani, Stephen Marsland, Isabel Castro, and Amal Punchihewa. Birdsong denoising using wavelets. *PloS one*, 11(1), 2016. (Cited on page 134)

Nirosha Priyadarshani, Stephen Marsland, and Isabel Castro. Automated birdsong recognition in complex acoustic environments: a review. *Journal of Avian Biology*, 49(5):jav–01447, 2018. (Cited on page 133)

Nirosha Priyadarshani, Stephen Marsland, Julius Juodakis, Isabel Castro, and Virginia Listanti. Wavelet filters for automated recognition of birdsong in long-time field recordings. *Methods in Ecology and Evolution*, 11(3):403–417, 2020. (Cited on page 134)

Lingjun Pu, Xu Chen, Jingdong Xu, and Xiaoming Fu. D2d fogging: An energy-efficient and incentive-aware task offloading framework via network-assisted d2d collaboration. *IEEE Journal on Selected Areas in Communications*, 34(12):3887–3901, 2016. (Cited on page 32)

Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016. (Cited on page 22)

Jorge Quesada, Paul Rodriguez, and Brendt Wohlberg. Separable dictionary learning for convolutional sparse coding via split updates. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4094–4098. IEEE, 2018. (Cited on page 25)

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016. (Cited on pages 5 and 27)

Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990. (Cited on page 30)

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018. (Cited on page 3)

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CPVR '16, pages 779–788. IEEE, jun 2016. ISBN 978-1-4673-8851-1. (Cited on page 3)

Haoyu Ren, Darko Anicic, and Thomas Runkler. Tinyol: Tinyml with online-learning on microcontrollers. *arXiv preprint arXiv:2103.08295*, 2021. (Cited on pages 31 and 32)

Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. Sbnet: Sparse blocks network for fast inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8711–8720, 2018. (Cited on page 25)

Luke E Richards, André Nguyen, Ryan Capps, Steven Forsyth, Cynthia Matuszek, and Edward Raff. Adversarial transfer attacks with unknown data and class overlap. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 13–24, 2021. (Cited on page 4)

Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2754–2761, 2013. (Cited on pages 5, 24, and 25)

Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978. (Cited on pages 89 and 90)

Martha Roseberry and Alberto Cano. Multi-label knn classifier with self adjusting memory for drifting data streams. In *Second International Workshop on Learning with Imbalanced Domains: Theory and Applications*, pages 23–37. PMLR, 2018. (Cited on page 61)

Kenneth V Rosenberg, Adriaan M Dokter, Peter J Blancher, John R Sauer, Adam C Smith, Paul A Smith, Jessica C Stanton, Arvind Panjabi, Laura Helft, Michael Parr, et al. Decline of the north american avifauna. *Science*, 366(6461):120–124, 2019. (Cited on page 133)

Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018. (Cited on page 90)

Zachary J Ruff, Damon B Lesmeister, Leila S Duchac, Bharath K Padmaraju, and Christopher M Sullivan. Automated identification of avian vocalizations with deep convolutional neural networks. *Remote Sensing in Ecology and Conservation*, 2019. (Cited on page 135)

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986. (Cited on page 51)

Manuele Rusci, Marco Fariselli, Alessandro Capotondi, and Luca Benini. Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers. In *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, pages 296–308. Springer, 2020. (Cited on pages 5 and 31)

Ramon Sanchez-Iborra and Antonio F Skarmeta. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020. (Cited on pages 4 and 31)

Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33, 2020. (Cited on page 26)

Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017. (Cited on page 25)

Simone Scardapane, Michele Scarpiniti, Enzo Baccarelli, and Aurelio Uncini. Why should we add early exits to neural networks? *Cognitive Computation*, 12(5):954–966, 2020. (Cited on page 28)

Heinrich Schwabe and Herrn Hofrath Schwabe. Sonnen – beobachtungen im jahre 1843. *Astronomische Nachrichten*, 21(15):234–235, 1844. doi: 10.1002/asna.18440211505. (Cited on page 147)

SEAL. Microsoft SEAL (release 3.4). `https://github.com/Microsoft/SEAL`, oct 2019. Microsoft Research, Redmond, WA. (Cited on page 94)

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014. (Cited on page 27)

Masayuki Senzaki, Jesse R Barber, Jennifer N Phillips, Neil H Carter, Caren B Cooper, Mark A Ditmer, Kurt M Fristrup, Christopher JW McClure, Daniel J Mennitt, Luke P Tyrrell, et al. Sensory pollutants alter bird phenology and fitness across a continent. *Nature*, 587(7835):605–609, 2020. (Cited on page 133)

Weiwei Shan, Minhao Yang, Tao Wang, Yicheng Lu, Hao Cai, Lixuan Zhu, Jiaming Xu, Chengjun Wu, Longxing Shi, and Jun Yang. A 510-nw wake-up keyword-spotting chip using serial-fft-based mfcc and binarized depthwise separable cnn in 28-nm cmos. *IEEE Journal of Solid-State Circuits*, 56(1):151–164, 2020. (Cited on page 23)

Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018. (Cited on page 23)

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):8815–8821, 2020. (Cited on page 28)

Cong Shi, Vasileios Lakafosis, Mostafa H. Ammar, and Ellen W. Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '12, pages 145–154, New York, NY, USA, 2012. ACM. (Cited on page 32)

Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. Cosmos: computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, pages 287–296, 2014. (Cited on page 33)

Dongjoo Shin, Jinmook Lee, Jinsu Lee, and Hoi-Jun Yoo. 14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 240–241. IEEE, 2017. (Cited on pages 22 and 23)

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. (Cited on page 3)

Gustavo Silva, Jorge Quesada, Paul Rodríguez, and Brendt Wohlberg. Fast convolutional sparse coding with separable filters. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6035–6039. IEEE, 2017. (Cited on page 25)

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. (Cited on page 99)

Michael R Smith, Tony Martinez, and Christophe Giraud-Carrier. An instance level analysis of data complexity. *Machine learning*, 95(2):225–256, 2014. (Cited on page 62)

Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS*, volume 1, page 2, 2014. (Cited on page 26)

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014. (Cited on pages 25 and 47)

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015. (Cited on page 51)

Rafael Stahl, Zhuoran Zhao, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Fully distributed deep learning inference on resource-constrained edge devices. In *International Conference on Embedded Computer Systems*, pages 77–90. Springer, 2019. (Cited on page 33)

RA Steenburgh, DA Biesecker, and GH Millward. From predicting solar activity to forecasting space weather: practical examples of research-to-operations and operations-to-research. In *Solar Origins of Space Weather and Space Climate*, pages 239–254. Springer, 2013. (Cited on page 145)

Dan Stowell, Michael D Wood, Hanna Pamuła, Yannis Stylianou, and Hervé Glotin. Automatic acoustic detection of birds through deep learning: the first bird audio detection challenge. *Methods in Ecology and Evolution*, 10(3):368–380, 2019. (Cited on page 141)

W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 377–382, New York, NY, USA, 2001. ACM. (Cited on page 30)

X Sun, Y Liu, JT Hoeksema, K Hayashi, and X Zhao. A new method for polar field interpolation. *Solar Physics*, 270(1):9–22, 2011. (Cited on page 145)

Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *Advances in Neural Information Processing Systems*, 33, 2020. (Cited on page 26)

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. (Cited on page 15)

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. (Cited on pages 3, 25, 28, and 51)

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. (Cited on pages 4 and 135)

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb 2017. (Cited on page 3)

Jihoon Tack, Sangwoo Mo, Jongheon Jeong, and Jinwoo Shin. Csi: Novelty detection via contrastive learning on distributionally shifted instances. In *Advances in Neural Information Processing Systems*, volume 33, pages 11839–11852, 2020. (Cited on page 3)

Chong Min John Tan and Mehul Motani. Dropnet: Reducing neural network complexity via iterative pruning. In *International Conference on Machine Learning*, pages 9356–9366. PMLR, 2020. (Cited on page 24)

Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020. (Cited on page 3)

Jie Tang, Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. Enabling deep learning on iot devices. *Computer*, 50(10):92–96, 2017a. (Cited on page 4)

Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Thirty-First AAAI conference on artificial intelligence*, 2017b. (Cited on page 27)

Zeyi Tao and Qun Li. esgd: Communication efficient distributed deep learning on the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018. (Cited on pages 5 and 33)

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016. (Cited on pages 28, 29, and 74)

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017. (Cited on pages 4, 5, and 33)

Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. In *Technologies for wireless computing*, pages 139–154. Springer, 1996. (Cited on page 18)

Ivan Tomek. An experiment with the edited nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):448–452, 1976a. (Cited on page 62)

Ivan Tomek. Two modifications of cnn. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772, 1976b. (Cited on page 62)

Gábor Tóth, Bart Van der Holst, and Zhenguang Huang. Obtaining potential field solutions with spherical harmonics and finite differences. *The Astrophysical Journal*, 732(2):102, 2011. (Cited on page 146)

Michael Towsey, Birgit Planitz, Alfredo Nantes, Jason Wimmer, and Paul Roe. A toolbox for animal call recognition. *Bioacoustics*, 21(2):107–125, 2012. (Cited on page 134)

Du Tran, Heng Wang, Lorenzo Torresani, and Matt Feiszli. Video classification with channel-separated convolutional networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5552–5561, 2019. (Cited on page 3)

Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Junichiro Kadomoto, Tomoki Miyata, Mototsugu Hamada, Tadahiro Kuroda, and Masato Motomura. Quest: A 7.49 tops multi-purpose log-quantized dnn inference engine stacked on 96mb 3d sram using inductive-coupling technology in 40nm cmos. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 216–218. IEEE, 2018. (Cited on page 23)

Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. In *5th International Conference on Learning Representations, ICLR 2017*, Toulon, France, April 24-26 2017. (Cited on page 24)

Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017. (Cited on pages 22 and 23)

Hossein Valavi, Peter J Ramadge, Eric Nestler, and Naveen Verma. A 64-tile 2.4-mb in-memory-computing cnn accelerator employing charge-domain compute. *IEEE Journal of Solid-State Circuits*, 54(6):1789–1799, 2019. (Cited on page 23)

Julio J Valdés, Ljubomir Nikolić, and Kenneth Tapping. Machine learning approaches for predicting the 10.7 cm radio flux from solar magnetogram data. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019. (Cited on page 148)

Julio J Valdés, Ljubomir Nikolić, Simone Disabato, and Manuel Roveri. A computational intelligence characterization of solar magnetograms. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020. (Cited on page 10)

Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010. (Cited on page 90)

Jan Van Leeuwen. On the construction of huffman trees. In *ICALP*, pages 382–410, 1976. (Cited on page 24)

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011. (Cited on page 26)

Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE transactions on neural networks and learning systems*, 30(2):326–342, 2018. (Cited on page 22)

Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2861–2865. IEEE, 2017. (Cited on page 27)

Marcus Venzke, Daniel Klisch, Philipp Kubik, Asad Ali, Jesper Dell Missier, and Volker Turau. Artificial neural networks for sensor data classification on small embedded systems. *arXiv preprint arXiv:2012.08403*, 2020. (Cited on pages 5 and 31)

Thomas Verelst and Tinne Tuytelaars. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. (Cited on page 29)

Fabio Vesperini, Leonardo Gabrielli, Emanuele Principi, and Stefano Squartini. A capsule neural networks based approach for bird audio detection. Technical report, DCASE2018 Challenge, September 2018. (Cited on pages 135, 142, and 143)

Mário Véstias, Rui Policarpo Duarte, José T de Sousa, and Horácio Neto. Parallel dot-products for deep learning on fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017. (Cited on page 22)

Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985. (Cited on page 30)

Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 315–332, 2018. (Cited on page 27)

Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys*, 52(2), May 2019a. (Cited on page 22)

Jiayun Wang, Yubei Chen, Rudrasis Chakraborty, and Stella X. Yu. Orthogonal convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020a. (Cited on page 25)

Joseph Wang, Kirill Trapeznikov, and Venkatesh Saligrama. Efficient learning by directed acyclic graph for resource constrained prediction. *arXiv preprint arXiv:1510.07609*, 2015. (Cited on page 29)

Min Wang, Baoyuan Liu, and Hassan Foroosh. Factorized convolutional neural networks. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 545–553, 2017a. (Cited on page 25)

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7686–7695, 2018a. (Cited on page 26)

Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 11–20, 2018b. (Cited on page 22)

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018c. (Cited on page 29)

XueSong Wang, Qi Kang, MengChu Zhou, Le Pan, and Abdullah Abusorrah. Multiscale drift detection test to enable fast learning in nonstationary environments. *IEEE Transactions on Cybernetics*, 2020b. (Cited on pages 30, 33, and 67)

Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019b. (Cited on page 22)

Zhisheng Wang, Jun Lin, and Zhongfeng Wang. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10): 2763–2775, 2017b. (Cited on page 23)

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018. (Cited on page 108)

Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. Tgpa: tile-grained pipeline architecture for low latency cnn inference. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8, 2018. (Cited on page 22)

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29:2074–2082, 2016. (Cited on page 25)

Paul N Whatmough, Sae Kyu Lee, David Brooks, and Gu-Yeon Wei. Dnn engine: A 28-nm timing-error tolerant sparse deep neural network processor for iot applications. *IEEE Journal of Solid-State Circuits*, 53(9):2722–2731, 2018. (Cited on pages 22 and 23)

Dennis L. Wilson. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-2(3):408–421, 1972. (Cited on page 62)

Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987. (Cited on page 148)

Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. Transfer learning with neural automl. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 8366–8375, 2018. (Cited on page 49)

Hongjia Wu, Jiao Zhang, Zhiping Cai, Qiang Ni, Tongqing Zhou, Jiaping Yu, Haiwen Chen, and Fang Liu. Resolving multi-task competition for constrained resources in dispersed computing: A bilateral matching game. *IEEE Internet of Things Journal*, 2021. (Cited on page 33)

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations*, 2018. (Cited on page 26)

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017. (Cited on page 127)

Yang Xiao. Ieee 802.11 n: enhancements for higher throughput in wireless lans. *IEEE Wireless Communications*, 12(6):82–91, 2005. (Cited on pages 118 and 130)

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017. (Cited on page 25)

Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. Bert-of-theseus: Compressing bert by progressive module replacing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7859–7869, 2020. (Cited on page 26)

Xiaowei Xu, Qing Lu, Tianchen Wang, Yu Hu, Chen Zhuo, Jinglan Liu, and Yiyu Shi. Efficient hardware implementation of cellular neural networks with incremental quantization and early exit. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 14(4):1–20, 2018. (Cited on page 28)

Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2369–2378, 2020. (Cited on pages 5, 28, and 29)

Linjie Yang and Qing Jin. Fracbits: Mixed precision quantization via fractional bit-widths. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):10612–10620, 2021. (Cited on page 27)

Zhe Yang, Sameer Al-Dahidi, Piero Baraldi, Enrico Zio, and Lorenzo Montelatici. A novel concept drift detection method for incremental learning in nonstationary environments. *IEEE transactions on neural networks and learning systems*, 31(1):309–320, 2019. (Cited on pages 26 and 30)

Zichao Yang, Marcin Moczulski, Misha Denil, Nando De Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1476–1483, 2015. (Cited on pages 24 and 25)

Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982. (Cited on page 90)

Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference*, pages 384–397. ACM, 2017. (Cited on page 87)

Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, and Kurt Keutzer. Hawq-v3: Dyadic neural network quantization. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11875–11886. PMLR, 18–24 Jul 2021. (Cited on page 27)

Seul-Ki Yeom, Philipp Seegerer, Sebastian Lapuschkin, Alexander Binder, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Pruning by explaining: A novel criterion for deep neural network pruning. *Pattern Recognition*, 115:107899, 2021. (Cited on page 24)

Shouyi Yin, Peng Ouyang, Jianxun Yang, Tianyi Lu, Xiudong Li, Leibo Liu, and Shaojun Wei. An energy-efficient reconfigurable processor for binary-and ternary-weight neural networks with flexible data bit width. *IEEE Journal of Solid-State Circuits*, 54(4):1120–1136, 2018. (Cited on page 23)

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014. (Cited on pages 52, 55, 83, 94, 135, 136, and 148)

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018. (Cited on page 24)

Zhe Yuan, Yongpan Liu, Jinshan Yue, Yixiong Yang, Jingyu Wang, Xiaoyu Feng, Jian Zhao, Xueqing Li, and Huazhong Yang. Sticker: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 55(2): 465–477, 2019. (Cited on page 23)

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019. (Cited on page 28)

Daniele Zambon, Cesare Alippi, and Lorenzo Livi. Concept drift and anomaly detection in graph streams. *IEEE transactions on neural networks and learning systems*, 29(11):5592–5605, 2018. (Cited on page 30)

Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3987–3995. PMLR, 06–11 Aug 2017. (Cited on page 30)

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015a. (Cited on pages 4 and 22)

Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*, pages 1984–1992, 2015b. (Cited on page 25)

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018a. (Cited on pages 24 and 25)

Xiaofan Zhang, Xinheng Liu, Anand Ramachandran, Chuanhao Zhuge, Shibin Tang, Peng Ouyang, Zuofu Cheng, Kyle Rupnow, and Deming Chen. High-performance video content recognition with long-term recurrent convolutional network for fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017. (Cited on page 22)

Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. Deep learning for environmentally robust speech recognition: An overview of recent developments. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(5):1–28, 2018b. (Cited on page 3)

Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018. (Cited on pages 5 and 33)

Yin Zheng, Bangsheng Tang, Wenkui Ding, and Hanning Zhou. A neural autoregressive approach to collaborative filtering. In *International Conference on Machine Learning*, pages 764–773. PMLR, 2016. (Cited on page 3)

Zibin Zheng, Yatao Yang, Xiangdong Niu, Hong-Ning Dai, and Yuren Zhou. Wide and deep convolutional neural networks for electricity-theft detection to secure smart grids. *IEEE Transactions on Industrial Informatics*, 14(4):1606–1615, 2017. (Cited on page 6)

Hao Zhou, Jose M Alvarez, and Fatih Porikli. Less is more: Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer, 2016a. (Cited on page 25)

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016b. (Cited on page 27)

Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33, 2020. (Cited on pages 5 and 29)

Yi Zhou, Yue Bai, Shuvra S Bhattacharyya, and Heikki Huttunen. Elastic neural networks for classification. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 251–255. IEEE, 2019. (Cited on pages 5 and 28)

Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979, 2020. (Cited on page 26)

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198. PMLR, 2017. (Cited on page 51)

Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 5(5):363–387, 2012. (Cited on page 148)

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018. (Cited on page 25)