UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Distributed DNN inference at the edge

Guo, X.

Publication date
2025
Document Version
Final published version

Link to publication

Citation for published version (APA):
Guo, X. (2025). *Distributed DNN inference at the edge*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:14 paź 2026

Distributed DNN Inference at the Edge

Xiaotian Guo

Distributed DNN Inference at the Edge

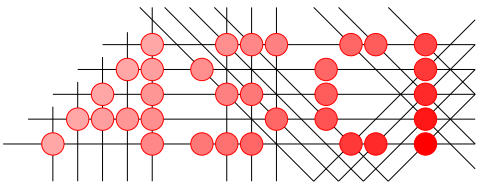
Xiaotian Guo

DISTRIBUTED DNN INFERENCE AT THE EDGE

XIAOTIAN GUO

UNIVERSITEIT VAN AMSTERDAM

Universiteit Leiden



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 463.
Copyright © 2024 Xiaotian Guo.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from the author.

Cover Design: from original art by Xiaotian Guo.
Thesis template: `classicthesis` by André Miede and Ivo Pletikosić.
Printed and bound by Proefschrift Specialist Printing
ISBN: 978-94-93391-89-5

DISTRIBUTED DNN INFERENCE AT THE EDGE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in
de Agnietenkapel
op woensdag 5 februari 2025, te 16:00 uur

door Xiaotian Guo
geboren te ANHUI

Promotiecommissie

Promotoren:	prof. dr. A.D. Pimentel	Universiteit van Amsterdam
	dr. T.P. Stefanov	Universiteit Leiden
Overige leden:	prof. dr. R.V. van Nieuwpoort	Universiteit Leiden
	prof. dr. D. Müller-Gritschneder	Technische Universität Wien
	prof. dr. ir. A. Iosup	Vrije Universiteit Amsterdam
	prof. dr. P. Grosso	Universiteit van Amsterdam
	dr. A. Pathania	Universiteit van Amsterdam
	dr. D. Sapra	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

This work is dedicated to my wife, parents, brother, sister-in-law, those I love, and those who love me.

CONTENTS

1 INTRODUCTION 1
- 1.1 AI Revolution 1
- 1.2 Edge Computing and Internet of Things 3
- 1.3 Deep Learning at the Edge 4
- 1.4 Why Distributed DNN Inference at the Edge? 6
- 1.5 Distributed DNN Inference at the Edge 8
- 1.6 Thesis Overview 13
  - 1.6.1 Origins 15
- 1.7 Author Publications 16
- 1.8 Source Code 16

2 BACKGROUND 17
- 2.1 DNN Model 17
  - 2.1.1 Convolutional Neural Network (CNN) 18
  - 2.1.2 Transformer 20
- 2.2 Partitioning Methods 22
- 2.3 Design Space Exploration 24
  - 2.3.1 Non-dominated Sorting Genetic Algorithm II 24
  - 2.3.2 Chromosome 25
  - 2.3.3 Fitness Function 26
- 2.4 Interoperability 31

I DISTRIBUTED DNN INFERENCE AT THE EDGE 33

3 CHAPTER 3 35
- 3.1 Introduction 36
- 3.2 Related Work 39
- 3.3 The AutoDiCE tool 41
  - 3.3.1 Overview 41
  - 3.3.2 Interface 44
  - 3.3.3 Front-end 46
  - 3.3.4 Back-end 47
- 3.4 Framework Evaluation 51
  - 3.4.1 Experimental Setup 51
  - 3.4.2 Efficiency of AutoDiCE and DSE Results 53

3.4.3 Varying the Number of Edge Devices 56
3.5 Discussion 57
3.6 Conclusions 59
4 CHAPTER 4 61
4.1 Introduction 62
4.2 Related work 63
4.3 Method 64
4.3.1 Fitness Functions 64
4.3.2 Multi-stage hierarchical DSE 64
4.4 Experimental Evaluation 67
4.4.1 Experimental setup 67
4.4.2 Experimental results 68
4.5 Conclusion 69

II ROBUSTNESS FOR DISTRIBUTED INFERENCE 71

5 CHAPTER 5 73
5.1 Introduction 74
5.2 Related Work 75
5.3 Background and Motivation 77
5.4 The RobustDiCE Method 79
5.4.1 Decentralized Computing Framework 80
5.4.2 Robust Partitioning 81
5.5 Evaluation of the RobustDiCE Method 85
5.5.1 Experimental Setup 85
5.5.2 Experimental Results 86
5.6 Conclusions 90
6 CHAPTER 6 91
6.1 Introduction 91
6.2 Related work 94
6.3 Robust Model Splitting 97
6.3.1 Motivational Example 98
6.3.2 Robust Model Splitting 100
6.4 Problem Formulation 101
6.5 The EASTER method 102
6.5.1 Partial Split Method for Transformers 102
6.5.2 Design Space Exploration 104
6.5.3 Multi-node Intermediate Representation 108
6.6 Evaluation of the EASTER method 109
6.6.1 Experimental Setup 109

6.6.2 DSE Results and Comparison 112
6.6.3 Robustness Verification Against Varying Failures 115
6.6.4 Distributed Inference 117
6.7 Conclusions 118
7 CONCLUSION 121
7.1 Answers to Challenges 123
7.2 Future work 127

BIBLIOGRAPHY 129

SUMMARY 137

SAMENVATTING 139

ACKNOWLEDGEMENTS 141

ACRONYMS

AI	Artificial Intelligence
CNN	Convolutional Neural Network
DNN	Deep Neural Network
CPU	Central Processing Unit
CONV	Convolution
CDC	Coded Distributed Computing
DSE	Design Space Exploration
EI	Edge Intelligence
FC	Fully Connected
GA	Genetic Algorithm
GPU	Graphical Processing Unit
HV	Hypervolume
IR	Intermediate Representation
IoT	Internet of Things
IPS	Images processed Per Second
LLM	large Language Model
LP	Layer Partitioning
MAC	Multiply Accumulate (arithmatic) operation
MOO	Multi-Objective Optimization
MOTPE	Multi-Objective Tree-structured Parzen Estimator

NSGA-II	Non-dominated Sorting Genetic Algorithm II
ONNX	Open Neural Network eXchange framework
PE	Processing Element
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SPE	Split Point Encoding
TPU	Tensor Processing Unit
UCB	Upper Confidence Bound
UCT	Upper Confidence bounds applied to Trees

1 INTRODUCTION

1.1 AI REVOLUTION

The artificial intelligence (AI) revolution represents a significant chapter in the evolution of computer science, transforming a broad range of sectors with applications in image classification, natural language processing (NLP), and more. The core of this transformative era is **deep learning (DL)**, a subset of AI that employs layered neural networks, enabling machines to analyze and learn from extensive datasets with a level of sophistication previously unattainable. Recent breakthroughs in AI applications like GPT-4, Copilot, etc., have demonstrated DL's potential in real-world scenarios. In computer vision, AI now exceeds human capabilities in object recognition and image classification. In NLP, AI achieves near-human comprehension and text generation, leading to advanced conversational agents. AI's application in autonomous systems, from self-driving cars to drones, demonstrates its ability to interpret and navigate complex environments autonomously. Additionally, the AI Index Report 2023 [1] highlights the remarkable growth in global AI private investment, which reached 91.9 billion in 2022, indicating an 18-fold increase since 2013. This underscores the escalating significance of AI across various sectors.

This revolution is characterized by exponential growth in AI research publications and the size of complex deep learning models. As shown in Figure 1.1, the last decade has seen an exponential increase in AI publications (red line), reflecting the burgeoning research activities in AI fields. Specifically, **deep neural networks (DNNs)** are at the forefront of the revolution in deep learning research, which draws inspiration from the neural architecture of the human brain. Driven by both academic and industrial sectors, the journey from simple perceptrons to Deep Neural Networks (DNNs) has been marked by a significant expansion in model size. DNNs have evolved from models with millions to billions of parameters. For in-

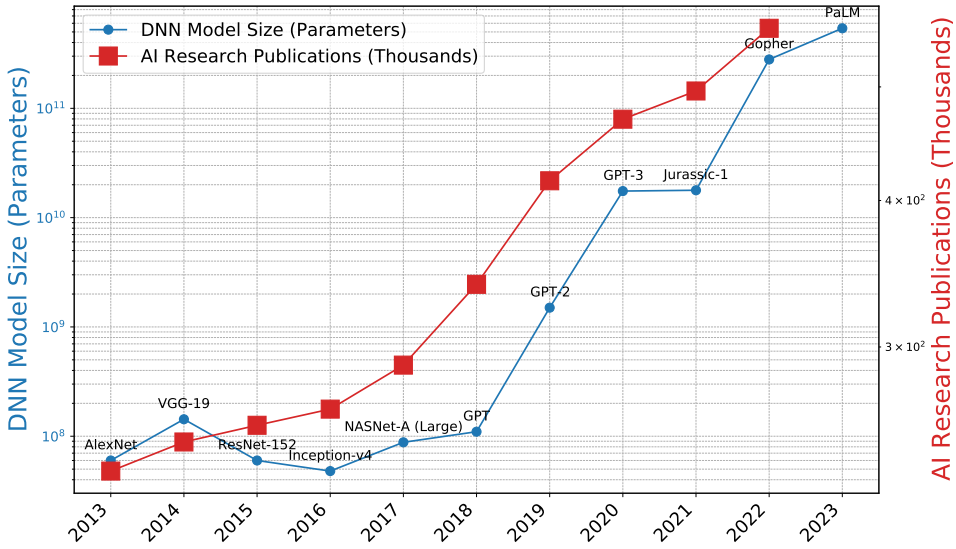


Figure 1.1: Growth of DNN Model Sizes and AI Research Publications (2013-2023)

stance, the shift from GPT-2 with 1.5 billion parameters in 2019 to Google's PaLM with 540 billion parameters in 2023 exemplifies the dramatic increase in model complexity. This trend towards larger models with exponentially growing parameters continues to push the boundaries of AI and deep learning research.

However, this rapid expansion brings a lot of challenges and opportunities. On the one hand, the demand for more powerful computational resources has never been higher. Training and deploying state-of-the-art deep learning models require significant computational power, often necessitating specialized hardware such as GPUs (Graphics Processing Units) or TPUs (Tensor Processing Units). On the other hand, the proliferation of advanced AI models also accentuates the need for advancements in computational efficiency, model optimization, and energy consumption. As models grow in size and complexity, the environmental impact of training these models becomes a concern, sparking efforts to develop more sustainable AI practices. Additionally, advanced AI in real-world applications often requires navigating a delicate balance between computational resources and user constraints, such as latency, memory, etc.

In essence, the AI revolution embodies a dynamic interplay between rapid technological growth and the ensuing challenges and opportunities. As the field continues to evolve, the urge for more efficient, powerful, and

sustainable AI systems remains at the forefront of technological innovation and research.

1.2 EDGE COMPUTING AND INTERNET OF THINGS

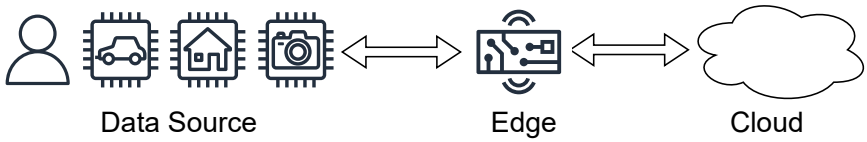


Figure 1.2: The Edge-Cloud Paradigm

The expansion of AI necessitates more powerful computational resources centralized in cloud computing infrastructures. As shown in Figure 1.2, "Cloud computing" is defined as a centralized Internet-based computing paradigm where large groups of centralized servers are networked in order to allow the sharing of computing services or resources. Thus, cloud computing is a largely centralized computing paradigm. The "Edge" here generally refers to a variety of networked devices with computing capacity placed anywhere along the path of data transmission between a data source and the cloud. In contrast to cloud computing, edge computing is a distributed computing paradigm in which services and computing resources are provided closer to the data sources and users.

The rapid growth of the Internet accelerates the expansion of the Internet of Things (IoT) devices, encompassing a wide array of devices ranging from sensors, laptops, and other software-equipped machines. These connected devices are capable of exchanging data not only among themselves but also with cloud systems. As highlighted by Toor et al. [2], it is expected that the total number of IoT devices connected will reach up to 60 billion by 2024. Billions of interconnected IoT devices generate significant volumes of data, providing the foundational data for the development, training, and refinement of AI models. IoT devices play a vital role in facilitating the collection, transmission, and in some cases, the processing of data between the cloud and various data sources.

While IoT devices offer massive data for training AI models within cloud servers, the challenges posed by the centralized cloud infrastructure, including latency, bandwidth limitations, and privacy concerns, intensify with the IoT device expansion. The challenges arising from centralized cloud infrastructures highlight the essential need to process AI workloads and deep

learning models closer to data sources or users—directly at the Edge. For instance, any delay in time-sensitive applications like autonomous vehicles can lead to significant, sometimes critical, outcomes. By adopting edge computing, issues such as latency, privacy concerns, and data storage pressures are mitigated, reducing dependencies on cloud-based infrastructures.

In total, the shift towards edge computing is becoming increasingly crucial for AI applications, and also provides a new landscape of AI deployment.

1.3 DEEP LEARNING AT THE EDGE

Deep learning, a specialized branch of artificial intelligence (AI), has seen remarkable development in research and industrial applications, including computer vision, natural language processing, the Internet of Things (IoT), autonomous systems, and other areas. Nowadays, deep neural networks (DNNs) are the front-runners of deep learning algorithms, known for their advanced capabilities, which enable machines to perform complex tasks with unprecedented efficiency and accuracy.

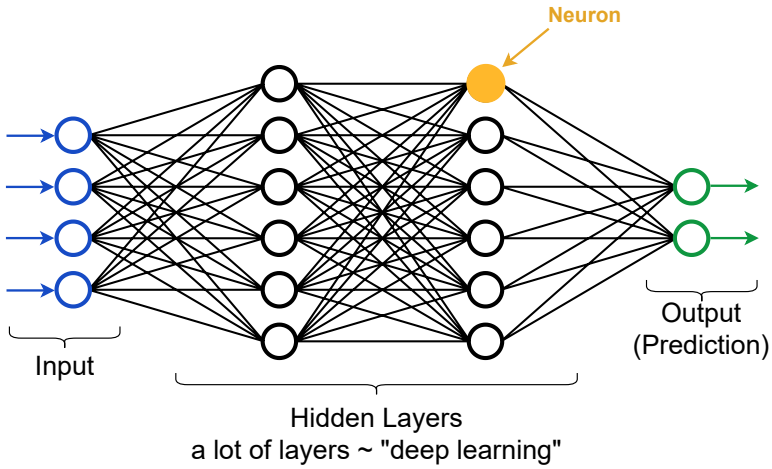


Figure 1.3: DNN Structure

A DNN (Figure 1.3) is structured into three primary parts: the input layer, hidden layers, and the output layer. The input layer processes initial raw data, such as image pixels or other data inputs, into input vectors. The hidden layers are multiple interconnected layers of numerous units for processing these input vectors. Ultimately, the output layer produces the final results of the DNN, such as generated text or image processing re-

sults. DNN models work in two main phases: **training** and **inference**. In the **training** phase, DNNs determine the values of the coefficients (weights and biases) within the hidden layers. This process involves learning from training datasets through a method known as gradient descent, which operates by backpropagation [3]. The basic units within the hidden layers, called neurons, contribute to the processing capability of its respective layer, and each hidden layer produces output data for the subsequent interconnected hidden layer. Each neuron in a DNN model executes a mathematical operation, such as convolution, dot product, rectifier activation [4], among others, leading to substantial computation and transformation within these neurons. In the **inference** phase, DNNs leverage these neurons to generate predictions based on input data. All coefficients are fixed and no longer change. These neurons work together to process, extract, and produce both high- and low-level representations or patterns from input data.

A key driver behind the advancement of DNNs is the substantial progress in computational hardware. Modern processors and GPUs have evolved to meet the demanding computational needs of DNNs, thereby accelerating the evolution of DNN models. The massive Deep Neural Network (DNN) computations can be executed through two main paradigms: cloud computing and edge computing. Each offers distinct advantages and addresses different requirements of user applications and the constraints of deployment environments.

Generally, the cloud server provides a powerful platform for performing DNN computations due to its huge computational resources, such as powerful high-performance graphic processor units (GPUs). The state-of-the-art DNNs typically contain hundreds or thousands of layers with numerous neurons, requiring a large amount of hardware resources for their training, deployment, or execution. DNN-based services are provided by these cloud servers in a centralized manner. Users initiate this service by sending requests, which include raw input data for the DNNs' input layers, to these cloud servers. Subsequently, the cloud servers allocate computational resources and data storage to handle these requests and return the results of DNNs' output layers (predictions) to the users. It is particularly well-suited for the training phase of DNNs, which requires handling vast datasets and performing extensive computations. Cloud servers enable the parallel processing of data and the deployment of sophisticated algorithms at a scale that would be impractical or prohibitively expensive for most local or on-premise solutions. The centralized nature of cloud computing also

simplifies the management of models and datasets, providing an efficient way to update, maintain, and scale AI applications.

However, relying solely on cloud servers for DNN computations can introduce challenges, including increased latency due to data transmission times between the client and the cloud, bandwidth constraints, and concerns regarding data privacy and security. Additionally, the continuous reliance on internet connectivity can pose limitations for applications that require real-time or near-real-time responses. Edge Computing, on the other hand, brings computation and data storage closer to the location where it is needed, aiming to reduce latency and bandwidth use. This approach is advantageous for deploying DNN models that require real-time inference, such as those used in autonomous vehicles, IoT devices, and real-time monitoring systems. By processing data locally on edge devices, response time can be significantly improved, which is critical for applications requiring immediate decision-making. Edge computing also enhances privacy and security, as sensitive data can be processed locally without being transmitted to a central server. For example, a network of IoT devices in smart healthcare systems within a hospital or a home setting, such as wearable health monitors, bedside monitors, and portable diagnostic devices, are equipped with sensors to collect vital signs and patient data in real time. By deploying DNN models directly onto these devices, the system can locally analyze data, make immediate health assessments, or predict medical events without the need to send or store sensitive patient data in centralized cloud servers, thus enhancing user privacy and data security.

In summary, edge intelligence (EI) combines edge computing with AI to process data closer to users. It facilitates the operation of DNN models in environments with limited or intermittent connectivity to the cloud. It enables decentralized data processing, making it possible for devices to reduce dependencies on cloud-based infrastructures. By doing so, issues such as latency, privacy concerns, and data storage pressures are mitigated in the DNN applications.

1.4 WHY DISTRIBUTED DNN INFERENCE AT THE EDGE?

While deploying Deep Neural Networks (DNNs) on edge devices offers several benefits, it also presents challenges like limited computational resources, high power consumption, and maintenance complexities. DNN inference is notably resource-intensive, and edge devices typically lack the requisite capabilities. These devices frequently have restricted processing

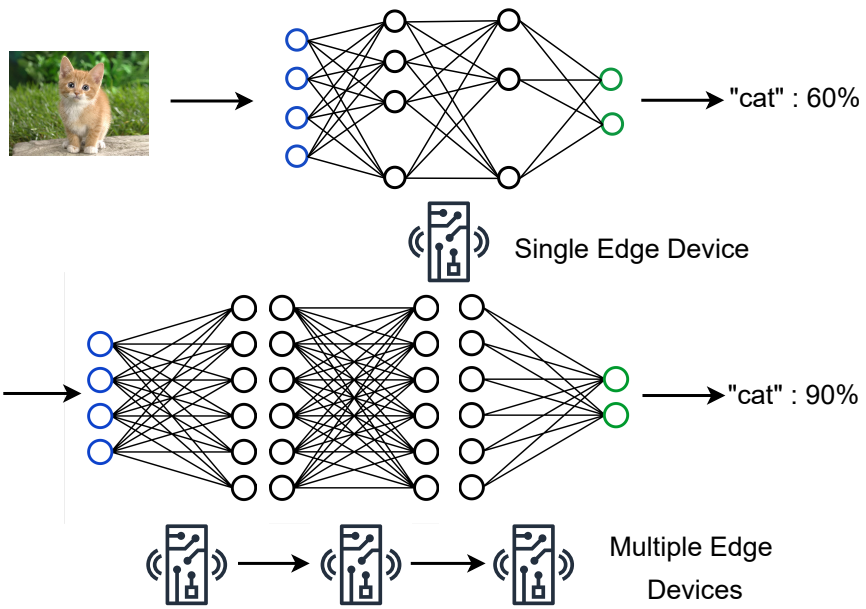


Figure 1.4: Why Distributed Inference at the Edge?

power and memory, creating obstacles in executing complex DNN models and necessitating compromises in terms of accuracy or functionality. Furthermore, the intensive computational demands of DNNs can rapidly drain batteries, particularly in applications requiring continuous operation, making power consumption a significant concern. Overcoming these limitations is crucial for the effective deployment of DNNs on edge devices. One approach is to use model compression methods such as pruning and quantization, albeit with possible impacts on accuracy. These methods focus on reducing the size of DNNs while trying to maintain their accuracy. Pruning and knowledge distillation help decrease memory and computational requirements, thus tailoring DNNs more suitably for edge devices. Quantization, by reducing numerical precision, further shrinks the model size and accelerates inference. As shown on the upper part of Figure 1.4, the model compression methods create a smaller DNN model appropriate for a single edge device. However, they incur significant retraining costs and would result in a potential accuracy drop. An alternative approach to address the challenge is to leverage all available resources along multiple edge devices to deploy and execute a large DNN by properly partitioning the DNN model and running each DNN partition on a separate edge device. The size of each DNN partition should match the limited energy, memory,

and compute resources of the edge device the partition runs on. Such an approach not only makes it possible to deploy large DNN models without the need of model compression, respectively without loss of accuracy, but it also resolves the aforementioned responsiveness and privacy issues because a cloud server is not involved in the DNN inference.

This thesis focuses on formulating strategies for the distributed deployment of Deep Neural Networks (DNNs) exclusively on edge devices. It aims to achieve an ideal equilibrium among throughput, energy consumption, and model accuracy while addressing the deployment limitations over distributed edge devices.

1.5 DISTRIBUTED DNN INFERENCE AT THE EDGE

In a distributed DNN inference scenario, each edge device is assigned a part of the DNN model to process. These devices either concurrently or sequentially perform their computations, adhering to the DNN's original operational sequence. The intermediate results from each device are collaboratively aggregated to produce the final output. This output is subsequently delivered back to the users, effectively leveraging the computational power and memory resources of multiple edge devices. However, implementing such distributed DNN computations across edge devices also presents a set of challenges, such as ensuring efficient coordination and communication between multiple devices, managing data consistency and synchronization, and minimizing the overhead associated with splitting and integrating model partitions.

The DNN distribution involves representing a DNN model into a computation graph and splitting the graph into a number of subgraphs/partitions. This process distributes DNN layers across different devices or divides the neurons within DNN layers among multiple devices. The subgraphs/partitions contain all neurons from the original graph and adhere to the computational graph's topological order.

Typically, a layer in a subgraph contains a portion of neurons from the original layer if that layer is distributed across multiple devices. To maintain the integrity of data flows within the computational graph and ensure alignment with the original DNN computation graph, the partitioned layer in the subgraph must be integrated with mechanisms for communication and synchronization. This is crucial for preserving the consistency and accuracy of the distributed inference process.

This problem leads us to our first research challenge:

CHL1: *How to flexibly and efficiently offload DNN models over multiple edge devices?*

Addressing this first challenge of distributing DNN models across multiple edge devices requires a comprehensive strategy that encompasses model parsing, partitioning, mapping, device management, and more. Therefore, we have developed AutoDiCE, a novel framework, detailed in Chapter 3, designed for efficiently exploring and automatically implementing a wide range of DNN partitions and mappings on multiple edge devices. This facilitates distributed DNN inference at the Edge. AutoDiCE starts with parsing and partitioning the DNN computational graph into smaller, manageable subgraphs. Following partitioning, AutoDiCE addresses the challenge of maintaining operational sequence and data integrity through its DNN mapping approach, which outlines the data flow between subgraphs. This process is distinct from single-device inference frameworks, as our framework supports distributed inference across multiple edge devices.

In AutoDiCE, we have introduced a method for parsing Deep Neural Network (DNN) models into a unified computational graph using the widely-used ONNX format, thereby enabling seamless connection with the PyTorch and TensorFlow frameworks used for training. In addition, AutoDiCE leverages dynamically generated C++ code, tailored for various hardware environments, to facilitate efficient model inference. The management of device resources, decentralized data communication, and systematic consistency checks ensure orderly execution and optimal data flow. Finally, AutoDiCE generates executable C++ files, thus enabling a cooperative computational execution for edge devices.

While Chapter 3 focuses on distributing DNN models at the Edge, it does not delve into the optimal distribution strategy. For instance, considering memory usage objectives may require evenly partitioned and balanced DNN weights across devices. Constraints on specific hardware with limited resources necessitate careful weight allocation to ensure successful execution. Finding an optimal distribution strategy that considers hardware constraints on edge devices is paramount.

Considering the heterogeneous computing architectures and varied available resources on each device, coupled with the complexity of a DNN model that can include hundreds of computational layers, each with different weight sizes, the design space of possible distributions of a DNN

model becomes exceedingly complex. For instance, distributing a DNN model with 100 layers across four devices introduces a staggering 4^{100} potential distribution strategies. This enormity of the design space underscores the substantial challenge in searching for an optimal distribution strategy that can efficiently accommodate the heterogeneity of devices. Balancing memory usage, energy consumption, and inference latency turns this into a multi-objective optimization problem. Efficient Design Space Exploration (DSE) methods are essential for navigating this vast design space, leading to our second research challenge:

CHL2: *How to perform an efficient DSE process to find optimal distribution strategies of DNN models at the Edge while considering multiple optimization objectives?*

To address the challenge of optimally distributing DNN models across heterogeneous devices, Chapter 4 introduces a novel multi-stage DSE method aimed at identifying Pareto optimal design points for the distribution of DNN models, particularly Convolutional Neural Networks (CNNs) that contain hundreds of layers. Given the vastness of potential distributions—for example, the myriad ways to distribute a CNN model's layers across multiple devices—finding an optimal solution within a limited number of search trials necessitates a tailored DSE approach. This DSE process is especially critical when considering various constraints such as memory usage, energy consumption, and inference throughput, where trade-offs among these objectives make it challenging to pinpoint a singular best solution, guiding the search towards identifying a Pareto front instead.

Typically, for Multi-Objective Optimization (MOO) problems, methods like Non-dominated Sorting Genetic Algorithm II (NSGA-II) [5] are employed for DSE. However, given the enormity of the search space in DNN distribution scenarios, NSGA-II would easily get stuck in local optima. To circumvent this limitation, we propose a NSGA-II-based DSE method that is specifically designed to overcome such pitfalls. This enhanced method leverages prior knowledge specific to distributed inference in a pipeline way to navigate the complex design space more effectively. Through this method, we aim to strike a balance between competing objectives, such as reducing memory and energy consumption while maximizing inference throughput, thus enabling efficient and effective deployment of distributed DNN inference systems.

It is crucial to acknowledge that all our aforementioned approaches and methods to address challenges **CHL1** and **CHL2** assume uninterrupted availability of edge devices, a condition that cannot always be guaranteed. Edge devices, especially those that are mobile or rely on low-power, short-distance communication technologies, may become temporarily inaccessible or suffer from failures, such as battery depletion. Given the inherent unreliability of such edge devices, that can easily change location and thereby disrupt availability, failure-resilient execution of distributed DNN-based applications throughout their active lifespan becomes essential. The transient unavailability of one or more edge devices during the distributed execution of DNN inference could lead to corrupted or inaccurate results, posing significant risks to user applications and potentially leading to severe consequences, such as automotive accidents or failures in smart care systems. This leads to our third research challenge:

CHL3: *How to ensure the robustness of DNN inference across multiple edge devices against possible failures or transient unavailability?*

To address this challenge, an insight that different neurons within each DNN layer contribute differently to the DNN inference accuracy has led us to the development of a novel method, called RobustDiCE, described in Chapter 5, for robust distribute DNN inference at the Edge. This method is inspired by the fact that every neuron within each layer of a DNN model has a different importance to the inference accuracy. By evaluating the importance of every neuron, neurons can be allocated across multiple devices in a manner that not only balances the computational load but also replicates critically important neurons. Such a strategy ensures that, despite potential device failures, the most crucially important neurons remain operational to the greatest possible extent, thereby enhancing the robustness of distributed DNN inference. In brief, RobustDiCE emphasizes two main aspects of robustness: system robustness, ensuring the continuity of DNN inference even when one or more edge devices malfunction, and model robustness, which aims at maintaining the DNN model inference accuracy despite the loss of some intermediate results due to device failures.

It is important to note that in RobustDiCE, a higher ratio of neuron replication correlates with increased DNN model inference accuracy in the face of potential device failures. However, this comes with the trade-off of

larger memory usage and more computational load per device due to the additional replicated neurons. With a DNN model potentially comprising hundreds of layers, determining an optimal replication ratio for each layer introduces the complex challenge of navigating through hundreds of possible ratio values for model distribution. Excessive replication contradicts the goal of minimizing the memory usage per device and reducing the energy consumption per device. Thus, it becomes crucial to identify an optimal set of replication ratios for each layer within the DNN model. The objective is to keep the replication ratio as low as possible (thereby ensuring that the memory usage and energy consumption on each device do not significantly increase due to neuron replication) while also providing a sufficient level of robustness against device failures. This leads us to our fourth and final research challenge:

CHL4: *How to determine an optimal set of neuron replication ratios for each layer within a distributed DNN model that maximizes inference accuracy in the face of potential edge device failures, while minimizing the impact on memory usage and computational load per device, thereby ensuring efficiency in terms of energy consumption and overall system performance?*

To address the outlined research challenge, we present a novel DSE method named EASTER in Chapter 6, specifically tailored for determining the optimal set of replication ratios. As similar replication ratio sets yield comparable design spaces, EASTER first divides the entire design space into smaller, manageable sub-spaces. Within each of these sub-spaces, EASTER estimates the expected reward for each optimization objective based on previously evaluated design points. This allows us identify and focus on the most promising or potentially promising sub-spaces for generating new design points. By concentrating the search and sampling efforts on these selected sub-spaces, EASTER significantly improves its search efficiency. This method strikes a balance between exploration of new possibilities and exploitation of known promising areas, effectively avoiding the common issue of getting trapped in local optima. Ultimately, the Pareto optimal points or solutions identified through this process represent a superior compromise between maintaining robustness to device failures and achieving operational efficiency in terms of computation efficiency and memory usage.

1.6 THESIS OVERVIEW

Figure 1.5 visualizes how this thesis is organized. Chapter 2 provides the basic background knowledge for the topics discussed throughout the rest of the thesis. This chapter introduces the basics of DNN models, the design space exploration methods applied in the distribution problem of DNN neurons across multiple devices, and interoperability requirements for DNN deployment at the Edge.

Following the background chapter, this thesis is organized into two major parts. The first part, consisting of two chapters, is focused on distributed inference at the Edge. Chapter 3 introduces a new fully automated tool for distributed deployment of CNN models over multiple resource-constrained devices at the Edge. It automates the splitting of a CNN model into a set of sub-models and automates code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices. Chapter 4 dives deeper into the optimal distribution strategies for the distributed deployment of CNN models, with a focus on optimizing the performance, memory usage, and energy consumption of the involved edge devices. It proposes an advanced DSE method with a new genetic encoding method for efficiently exploring the pipe-lined distributions of CNN models at the Edge to improve (i.e., reduce) per-device energy consumption, reduce per-device memory usage, and improve system inference throughput (under certain conditions) as well. This chapter depends heavily on the AutoDiCE framework presented in Chapter 3, to derive actual implementations of the resulting Pareto solutions considering the three aforementioned optimization objectives.

The second part of the thesis contains two chapters and is focused on the robustness of distributed DNN inference. The work presented in both of these chapters builds upon the implementation of the AutoDiCE framework (the first part of the thesis). However, the work in the two chapters in the first part assumes continuous availability of all involved edge devices, which cannot be guaranteed at all times because an edge device could fail or become temporarily unreachable, e.g., due to an unstable connection, a depleted battery, etc. Thus, Chapter 5 proposes the RobustDiCE method to enhance the robustness of distributed inference against possible device failures by using a balanced dispersion of critical neurons over various devices by assessing the importance of neurons within DNN layers.

The RobustDiCE method in Chapter 5 replicates crucial neurons across multiple devices to allow for robust distributed DNN inference. However,

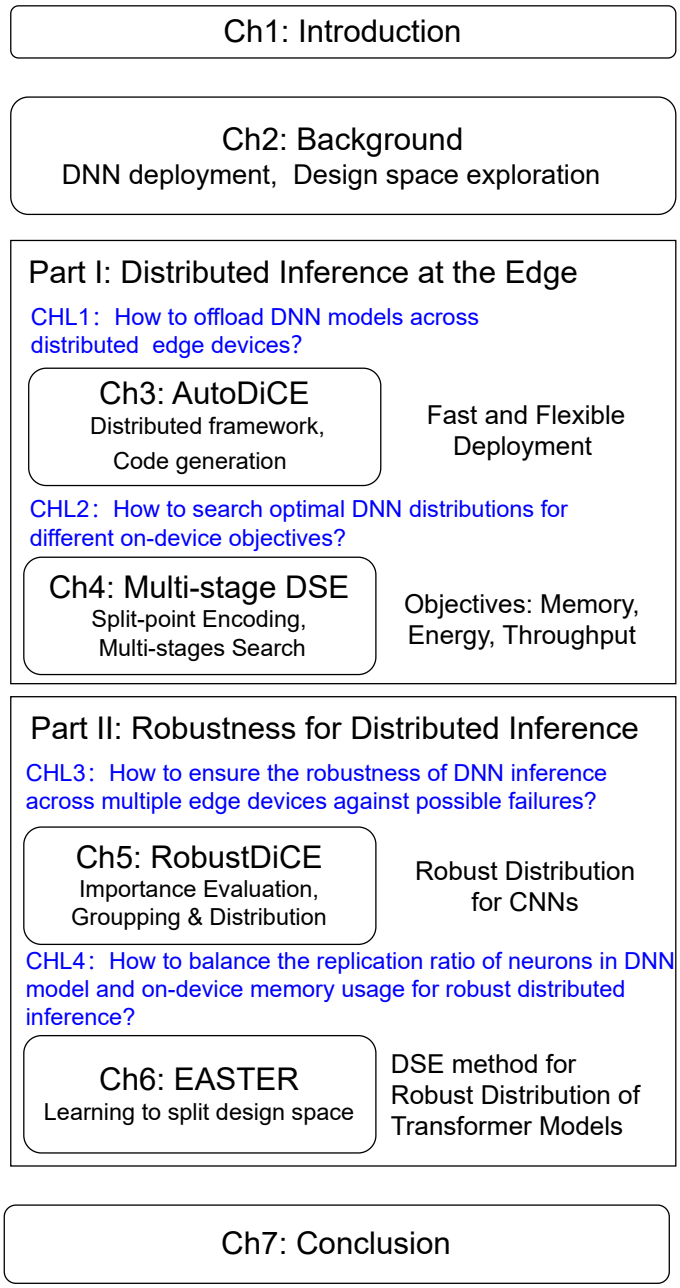


Figure 1.5: Thesis Organisation, including chapters, keywords, and research questions

the actual determination of the appropriate replication proportion of neurons within DNN layers is left open and remains unaddressed. To tackle

this challenge, Chapter 6 introduces the EASTER method to efficiently search for optimal replication ratios for each layer in the DNN models designated for distribution, aiming to identify the most effective replication strategy against device failures that maintains the model performance of distributed inference while optimizing resource utilization. More specifically, the type of DNN models studied in Chapter 6 are the popular but very large transformer models known from tools like ChatGPT and Copilot.

These four research chapters contain our core contributions to distributed inference at the Edge. In the final chapter of the thesis, Chapter 7, we reflect on the research challenges and present some ideas for future work.

1.6.1 *Origins*

Listed below are the author's contributions and papers on which each of the research chapters is based. The next section enumerates the full list of the author's publications.

PART I

ch.3: *"Automated exploration and implementation of distributed CNN inference at the Edge"* [P1]

ch.4: *"Hierarchical design space exploration for distributed CNN inference at the Edge"* [P2]

PART II

ch.5 *"RobustDiCE: Robust and Distributed CNN Inference at the Edge"* [P3]

ch.6 *"EASTER: learning to split transformers robustly at the Edge"* [P4, P5]

For papers [P1–P5], the author of this thesis is the principal author and performed all of the data analysis, software development, experimental set-up, validation, and writing of the original draft.

1.7 AUTHOR PUBLICATIONS

[P1] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. "Automated Exploration and Implementation of Distributed CNN Inference at the Edge." In: *IEEE Internet of Things Journal* 10.7 (2023), pp. 5843–5858. DOI: 10.1109/JIOT.2023.3237572.

[P2] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. "Hierarchical design space exploration for distributed CNN inference at the edge." In: *3rd Workshop on IoT, Edge and Mobile for Embedded Machine Learning (ITEM 2022), part of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2022, pp. 545–556. DOI: 10.1007/978-3-031-23618-1_36.

[P3] Xiaotian Guo, Quan Jiang, Andy D. Pimentel, and Todor Stefanov. "RobustDiCE: Robust and Distributed CNN Inference at the Edge." In: *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2024, pp. 26–31. DOI: 10.1109/ASP-DAC58780.2024.10473970.

[P4] Xiaotian Guo, Quan Jiang, Andy D. Pimentel, and Todor Stefanov. "Model and System Robustness in Distributed CNN Inference at the Edge." In: *Integration, the VLSI Journal*. DOI: 10.1016/j.vlsi.2024.102299.

[P5] Xiaotian Guo, Quan Jiang, Yixian Shen, Andy D. Pimentel, and Todor Stefanov. "EASTER: learning to split transformers robustly at the Edge." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2024). DOI: 10.1109/TCAD.2024.3438995.

1.8 SOURCE CODE

- The AutoDiCE implementation and the hierarchical DSE script is available at: https://github.com/parrotsky/AutoDiCE
- The RobustDiCE implementation as described in Chapter 5 is available at: https://github.com/parrotsky/RobustDiCE
- The EASTER implementation script for the DSE process in Chapter 6 is available at: https://github.com/parrotsky/EASTER

2

BACKGROUND

In this chapter, we provide an overview of essential background knowledge for this thesis, covering several core areas: basics of DNNs, design space exploration (DSE), and interoperability requirements for DNN deployment at the Edge. By discussing these key terminologies and fundamental methods, we set the stage for deeper discussions and explorations of specialized topics in subsequent chapters.

2.1 DNN MODEL

A typical DNN contains numerous neurons, leading to a significant number of parameters, collectively known as model weights or coefficients. These coefficients are determined during the model's training phase through exposure to training data, utilizing the back-propagation algorithm and gradient descent technique [3]. Assuming the correct output labels for each input are known, the training process starts with a forward pass through all layers in the DNN model to produce a predicted output. This output is then utilized to calculate the error, which is the discrepancy between the predicted output and the actual output, followed by computing the gradients of this error for each weight in the neural network. These gradients are propagated backward from the output layer towards the input layer of the network. This backward movement ensures that each weight in the network is adjusted in a way that minimizes the overall error. This gradient descent technique allows for precise weight updates, improving the model's accuracy and performance with each iteration of training.

Once a neural network is adequately trained with sufficient data after enough iterations, it can be utilized for inference. During the inference phase, the pre-trained model applies its learned coefficients to new input data and makes predictions or decisions for its designated tasks, such as image classification, object detection, etc.

Next, we will discuss two commonly used DNNs: CNN [6] and Transformer [7], along with the operations within their layers.

2.1.1 *Convolutional Neural Network (CNN)*

Convolutional Neural Networks (CNNs) are a fundamental class of deep neural networks renowned for their effectiveness in handling vision-related tasks such as image classification, object recognition, and detection. A CNN model consists of various layers including convolution layers, pooling layers, non-linear activation layers, and fully connected layers, etc. Typically, the convolution layers and fully connected layers are the most computationally and memory-intensive parts of a CNN model. In Figure 2.1, we illustrate the computation of a convolution layer and fully connected layer in a uniform way from a neuron perspective by considering four neurons per layer as an example.

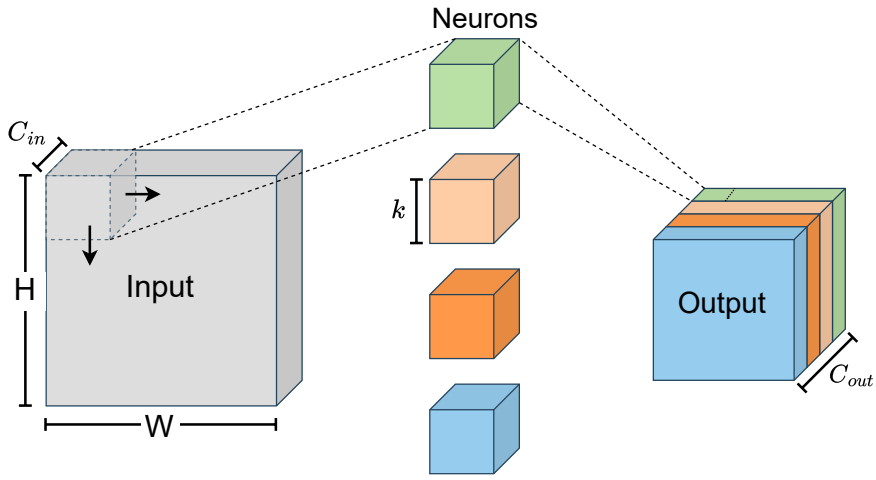


Figure 2.1: Neuron Computation

Convolution Layers: Neurons in convolution layers specialize in capturing spatial hierarchies and textures by applying convolution operations to the input images. After the convolution operations, neurons typically apply a non-linear activation function, such as ReLU [4], to introduce non-linearity into the model. This allows the CNN model to learn more complex patterns and relationships within the data while also helping to mitigate the vanishing gradient problem during back-propagation [4]. Figure 2.1 shows how each neuron performs a basic convolution operation as a filter (also called kernel). In this particular example of a convolution layer, each of the

four neurons corresponds to a unique convolutional filter (depicted as the small cubes in Figure 2.1). These filters contain their own sets of weights, structured in a $k \times k \times C_{in}$ format, where k represents the filter's size, and C_{in} denotes the number of input channels that the filter processes.

During the convolution operation, each filter moves systematically over the input data — such as an image — applying its weights through a dot product operation at every position it covers. This process effectively extracts features from the input by highlighting specific patterns or textures, depending on the filter's weights.

The movement of each filter is governed by a stride value, S, which dictates how many units the filter moves to the right after each operation. Once the filter reaches the end of the input width, it returns to the leftmost position and moves down by one stride, after which it continues the process until the entire input has been covered. Padding is utilized to ensure that the filters can be applied to the edges of the input. We use P to represent the padding size. By adding P zeros (or other values) around the borders of the input, padding allows the filters to operate on the edges without reducing the size of the output. This process is crucial for maintaining the dimensional consistency of the output across different inputs.

The convolutional filters are designed to operate on all input channels (C_{in}), where each channel is a separate layer of height H and width W within the input data (represented as the large gray cube in Figure 2.1). As each filter slides over the input data, it aggregates information across the height and width dimensions to produce a new output channel or feature map. Thus, the number of output channels (C_{out}) in the resultant feature map is equivalent to the number of convolutional filters used in the layer. In this example, with four filters, the output feature map (C_{out}) will also have four channels, each channel representing the output from one of the convolutional filters.

Fully Connected Layers: Each neuron in a fully connected layer is connected to all inputs from the previous layer, integrating the learned features comprehensively. A fully connected layer could be seen as a special case of a convolution layer where each filter has the same shape as the shape of the input, i.e., $k \times k \times C_{in} = H \times W \times C_{in}$. For the fully connected layers after the first fully connected layer in a CNN, the following relation holds $k = H = W = 1$. Every filter/neuron produces only one value, thus if the number of filters/neurons is C_{out} then the output has the shape $1 \times 1 \times C_{out}$.

As each filter/neuron has a distinct set of weight values, corresponding to the number of inputs, this leads to a significant total number of parame-

ters ($C_{in} \times C_{out}$) within the layer. Due to the number of parameters and the heavy computational complexity in fully connected layers, modern CNN models, such as ResNet [8], MobileNet [9], and YoLo [10], have evolved to reduce or eliminate fully connected layers. This makes the neural networks more efficient and beneficial for deployment on devices with limited computational capabilities, facilitating faster and more resource-efficient model execution without compromising accuracy.

2.1.2 *Transformer*

Following the introduction of the CNN model, it is crucial to discuss another significant model in the realm of deep learning: the Transformer model [7]. This model has gained substantial recognition, particularly for its groundbreaking successes in both computer vision (CV) [11] and natural language processing (NLP) [12].

At the beginning of a Transformer model, the input sequence is generated by mapping the raw input data — whether words in text or other types of data features — into a dense, fixed-size ($L \times d_{model}$) vector representation. The length of the input sequence L is determined by the total number of tokens it contains. In the context of natural language processing (NLP), each token typically corresponds to a word or a subword unit. In image processing tasks, on the other hand, a token may represent a specific block of pixels. The size of this fixed vector is defined by the model dimensionality d_{model}, which determines the capacity of the model to learn and represent information. A larger model dimensionality can increase the model's ability to capture complex patterns but also raises the computational requirements and model complexity. Every token in the input sequence is represented by a vector of this fixed dimensionality, ensuring uniformity in the input layer that feeds into subsequent modules of the model, like positional encoding and the encoder layers, etc.

As illustrated in Figure 2.2, the Transformer model structure is designed to transform one input sequence into another using two main components: an encoder and a decoder. The encoder is on the left and the decoder is on the right side of Figure 2.2. Denoted by "N×" in the figure, the encoder, and decoder are composed of N stacked transformer modules. Position-based encoding is another critical feature of the Transformer model, adding unique positional information to the input sequence to preserve the order of the sequence. Unlike recurrent neural networks (RNNs) [13], that process data sequentially and thus inherently understand the order of tokens in the

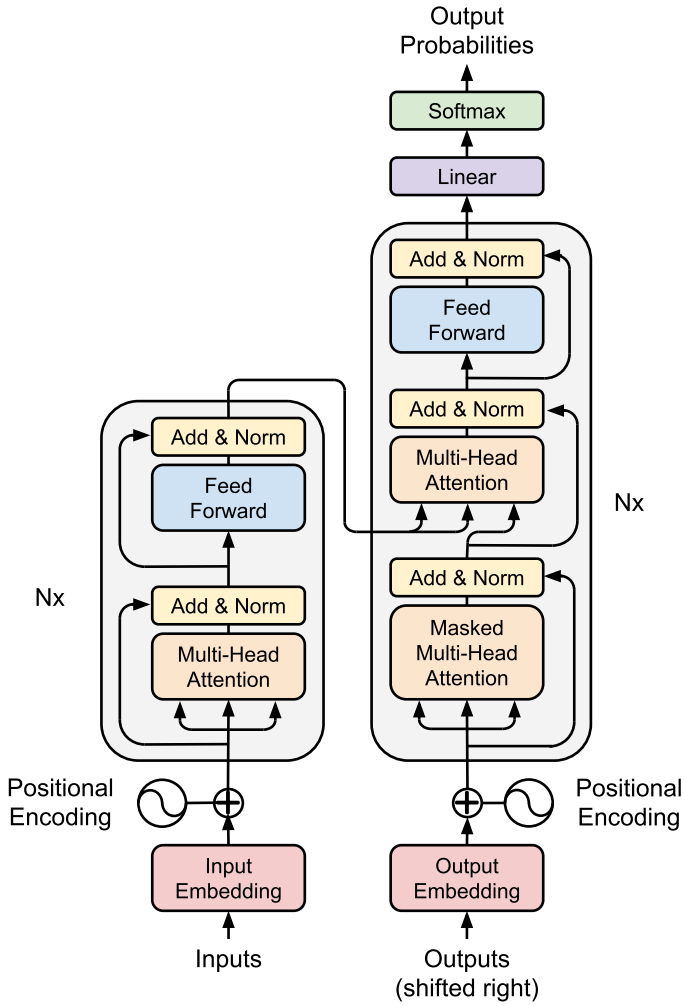


Figure 2.2: Transformer Model Structure By Vaswani et al. [7]

input sequence, Transformers add information about the order of tokens (words or the location of input image pixels), ensuring that the token's order contributes to the interpretation of the sequence. This enables the model to perform tasks like language translation, image recognition, or other tasks.

The encoder takes the input sequence and converts it into a continuous representation that retains all the information needed to generate the output. This representation is often thought of as a set of n-dimensional vectors ($L \times d_{model}$), where each vector corresponds to a token in the input

sequence but encapsulates information from the entire sequence. The decoder then takes this dense representation and step-by-step generates the outputs. This process starts with a first start-of-sequence (<sos>) token as the 'Outputs'. The decoder, through its layers, processes this token while also referencing the encoder's outputs to make context-aware predictions for the next output token in the sequence. The output from the decoder's last layer for each token is a probability distribution over all possible tokens in the model's vocabulary. From this distribution, the token with the highest probability is typically chosen (or sampled, depending on the specific application and settings), and this token is then used as the next input to the decoder. This process of generating a token and then feeding it back into the decoder continues until the decoder produces an end-of-sequence (<eos>) token, which indicates that the generation process should stop.

The encoder and decoder consist mainly of Multi-Head Attention and Feed Forward network modules. The **Multi-Head Attention** in the encoder lets each position in the encoder attend to all positions in the previous layer of the encoder. This self-attention mechanism helps the encoder look at other words or image pixels in the input sentence to better understand the context and meaning of each token. However, the Multi-Head Attention in the decoder is masked to ensure that the position can only attend to earlier positions in the output. This prevents the decoder from seeing future tokens in the sequence it is generating. For the second Encoder-Decoder multi-head attention, each position in the decoder can attend to all positions in the encoder, allowing the decoder to access the complete input sequence. This is crucial for tasks like translation, where understanding the entire input is necessary to generate the correct output. The **Feed Forward network** consists of two linear transformations with a ReLU activation in between, introducing non-linear capabilities to the model, and allowing it to learn more complex representations.

The combination of Multi-Head Attention and Feed Forward network allows Transformers to perform very well across a range of tasks, from language translation and text generalization to image processing and beyond, leveraging their ability to capture and utilize complex patterns in data.

2.2 PARTITIONING METHODS

Partitioning a DNN model, particularly when targeting inference with a batch size of one, involves dividing the model into smaller, more manageable segments. These segments can then be distributed and processed

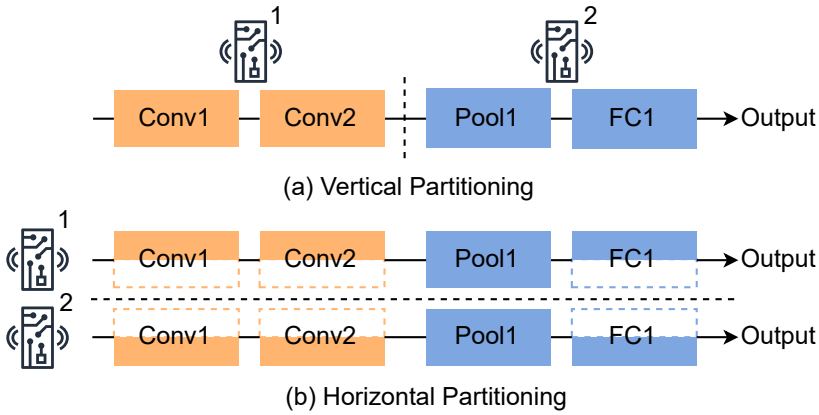


Figure 2.3: Partitioning Methods

across multiple devices, thereby optimizing resource usage and enhancing computational efficiency at the Edge. To achieve these distribution benefits of DNN deployment, two common approaches are typically utilized: *vertical* and *horizontal* partitioning.

Vertical partitioning (e.g., [14–16]), illustrated in Figure 2.3(a), involves dividing the entire model into several segments that can be executed on different devices. For example, the initial layers of a DNN model (Conv1 and Conv2) could be run on one device, while the deeper layers (Pool1 and FC1) are processed on another. This approach not only balances the computational load across multiple devices but also enhances the system throughput compared to processing large models on a single device.

Horizontal partitioning (e.g., [17–22]), illustrated in Figure 2.3(b), distributes the workload of one or all DNN layers among multiple edge devices in a layer-wise manner. This distribution strategy takes into account the varying computational demands of different layers within the DNN, allocating them across the devices strategically. For example, more computational-intensive convolution layers or more memory-intensive fully connected layers might be distributed across multiple edge devices to leverage combined computational resources. Less demanding layers, such as pooling layers, can be managed effectively by individual devices. These layers can be replicated across multiple devices for redundancy and faster access, but they do not require partitioning across different devices due to their lower computational or memory demands.

Although the discussed partitioning methods are exemplified through CNN models, they are equally applicable to other models like Transformers.

For example, fully connected layers in Transformers can also be partitioned either vertically or horizontally.

In summary, all of the aforementioned partitioning methods reduce the required memory, energy consumption, or computation resources per device when a DNN model is deployed for distributed inference on multiple devices [23]. At the same time, these methods may slow down the inference due to communication and synchronization overheads that are inevitable in distributed DNN inference across multiple devices. By enabling distributed inference, these methods help to overcome the limitations of individual devices, ensuring that even resource-constrained devices at the Edge can participate in complex computational tasks. This not only improves the efficiency and scalability of DNN deployments but also broadens the applicability of deep learning technologies across various sectors.

2.3 DESIGN SPACE EXPLORATION

Identifying optimal distribution solutions for Deep Neural Network (DNN) models, particularly for their deployment across multiple edge devices, is crucial for enhancing computational efficiency, reducing memory usage, or saving energy consumption in resource-constrained environments. Central to this effort is Design Space Exploration (DSE), which systematically evaluates various distribution configurations to identify solutions that effectively balance computational efficiency, power consumption, memory usage, and latency. Among the various methods [24] used in DSE, the Non-dominated Sorting Genetic Algorithm II (NSGA-II) is particularly noteworthy. This particular Genetic Algorithm (GA) is extensively used to adeptly manage the trade-offs between the aforementioned objectives (i.e., power consumption, memory usage, latency, etc.), exploring a wide array of potential solutions to find the most efficient and practical distributions.

2.3.1 *Non-dominated Sorting Genetic Algorithm II*

Non-dominated Sorting Genetic Algorithm II (NSGA-II) [5] is a well-known heuristic method recognized for its effectiveness in multi-objective optimization (MOO). NSGA-II operates by evolving a population of candidate solutions towards higher quality solutions, leveraging its unique components: the **chromosome** and the **fitness function**.

The evolution process in NSGA-II is iterative and repeats for a predetermined number of generations or until a convergence criterion is satisfied.

It initializes a random population of candidate solutions, represented as chromosomes. These chromosomes are evaluated using fitness functions to obtain the target objectives (memory, energy, latency, etc.). The NSGA-II algorithm utilizes non-dominated sorting to categorize the population into different fronts based on Pareto dominance. The first front, known as the Pareto front, contains a set of non-dominated solutions in multi-objective optimization that are considered optimal because improving one objective would require worsening another. Then, it selects individuals based on their performance (objective values) and how well they are spread out (crowding distance [25]) within their front. This ensures that the selected population is both high-performing and diverse. Genetic operators such as mutation and crossover are then applied to the selected parents to produce offspring. The resulting pool of parents and offspring is then sorted by non-dominated sorting and crowding distance. The top solutions are selected and used in the next iteration of the NSGA-II-based search. This cycle of generation and selection is continuously repeated until the termination condition is met.

Through this structured generation and selection approach, NSGA-II is highly effective in navigating complex solution spaces, proving essential for our search for optimal DNN distribution solutions.

2.3.2 *Chromosome*

In NSGA-II, a chromosome is the genetic representation of a solution, typically represented as a series of parameters known as genes. In distributed DNN inference scenarios, each chromosome corresponds to a potential DNN mapping solution that specifies the distribution of DNN layers or computations across multiple edge devices. A simple mapping chromosome for an eight-layer DNN model is visualized in Figure 2.4(a), where the chromosome consists of eight genes, each specifying which processing element is responsible for the computations of a particular layer. For example, the first gene in the chromosome means that layer l_1 is mapped on processing element PE_1, which could, e.g., be a CPU or GPU processor inside a particular edge device.

Once the most promising chromosomes are selected, the next iteration of the NSGA-II-based search creates its new population through genetic operators, namely recombination (or crossover) and mutation. During recombination, segments of chromosomes from two parents are exchanged to form new chromosomes, while mutations involve random changes to genes within a chromosome to maintain genetic diversity within the population.

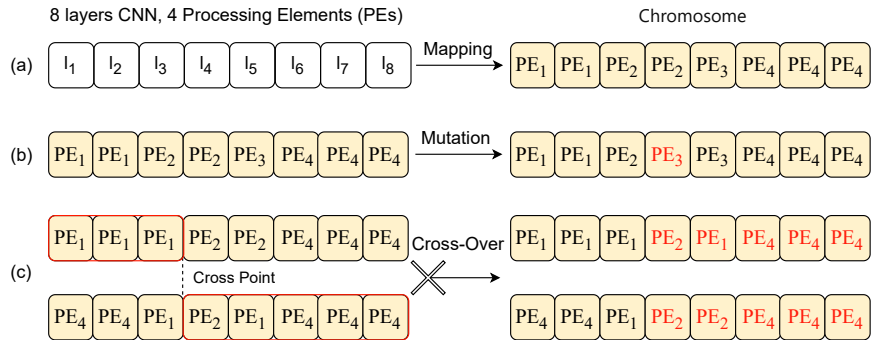


Figure 2.4: Chromosome & Genetic Operators

In this thesis, we use a standard two-parent crossover and a single-gene mutation as proposed in [26]. These operations are depicted in Figure 2.4(b) and (c). For instance, a mutation in the fourth gene of a chromosome from PE_2 to PE_3 introduces variability. Each chromosome is assigned a random mutation probability. Similarly, the single-point crossover operation shown in Figure 2.4(c) combines segments of chromosomes from two elite-selected parents at a random crossover point, potentially creating superior offspring by mixing their genetic material.

2.3.3 *Fitness Function*

The fitness function, another crucial component of NSGA-II, evaluates the performance or specific objectives of different solutions represented by chromosomes. By evaluating the fitness of each solution, the NSGA-II search algorithm can effectively steer toward solutions that optimize specific objectives. In our work, two approaches are employed to assess the fitness of solutions: the use of analytical models and the direct evaluation via measurements of actual physical systems.

2.3.3.1 *Analytical Model-based Fitness Function*

The first approach involves using analytical models to approximate the system throughput, memory usage, and energy consumption for a given DNN mapping. In the below discussion, we assume a vertical partitioning of a DNN. Furthermore, we use t_{l_j}, M_{l_j}, E_{l_j} to represent the execution time, the memory usage, and the energy consumption of layer l_j in a DNN model, respectively. A vertically partitioned mapping $\mathbf{x}$ of a DNN model is denoted as $\mathbf{x} = [x_1, x_2, \cdots, x_L]$, where L is the number of layers in the DNN model

and $x_j = PE_i$ means that layer l_j is mapped on processing element PE_i, which could, e.g., be a CPU or GPU inside an edge device. For a given mapping $\mathbf{x}$, the three objectives of the distributed system can be computed as follows.

System Throughput: The overall system throughput T_{system} is defined as the images processed per second (IPS) over multiple PEs:

$$T_{system} = \frac{1}{\max_{1 \leqslant i \leqslant N} (t^i)}; \; t^i = t^i_{comp} + t^i_{comm} \; ; \; t^i_{comp} = \sum_{\forall j: 1 \leqslant j \leqslant L \wedge x_j = PE_i} t_{l_j}$$

where t^i is the time to process one image on PE_i, N is the total number of deployed PEs in the distributed system. Specifically, t^i_{comp} and t^i_{comm} are the time needed for computation and data communication related to PE_i, respectively. We assume that the size of input images is already determined and the input and output tensors of every DNN layer are also fixed. Then, we can estimate the total number of operations in every layer and the total size of communicated data related to PE_i. The execution time t_{l_j} is estimated through the number of multiply-accumulate operations (MACs) and the processor frequency of PE_i. Assuming the processor executes one MAC operation per nc number of cycles, the execution time t_{l_j} (in seconds) of DNN layer l_j on processor PE_i can be calculated as:

$$t_{l_j} = \frac{MAC_{l_j}}{Freq_{PE_i}} \times nc$$

where MAC_{l_j} is the total number of MAC operations performed in l_j and $Freq_{PE_i}$ is the clock frequency of PE_i. The total number of MAC operations in a layer depends on the type of the layer. As an example, we give the MACs calculation of three specific types of CNN layers, namely convolution, fully connected, and pooling layers, that are essential for understanding the computational complexity of different CNN layers.

$$MAC_{conv} = H_{out} \times W_{out} \times k^2 \times C_{in} \times C_{out}$$

$$MAC_{fc} = C_{in} \times C_{out}$$

$$MAC_{pool} = H_{out} \times W_{out} \times k^2 \times C_{out}$$

where k is the kernel/filter size, C_{in} is the number of input channels, C_{out} is the number of output channels, H_{out} and W_{out} are the width and height of output tensors of the layer. Additionally, the proper approximation for communication time t^i_{comm} of PE_i depends on data movements and involves intra-node shared memory communication, intra-node communication between CPU and GPU, or inter-node communication over the network. For instance, by adding the total memory size of the communication

buffers involved and dividing it by the communication bandwidth between PEs, we can calculate the estimated time for communication t^i_{comm}.

Memory: Every PE_i allocates memory M^i which consists of three parts, namely memory for DNN coefficients (i.e. weights, bias, and parameters), memory for output buffers to store intermediate results of layers, and memory for input buffers of some layers to receive data from other PEs:

$$M^i = \sum_{\forall j:1 \leqslant j \leqslant L \wedge x_j = PE_i} (M^j_{coeffs} + M^j_{outbuffs} + M^j_{inbuffs})$$

where M^j_{coeffs}, $M^j_{outbuffs}$, and $M^j_{inbuffs}$ denote the sizes of the aforementioned memory parts associated with layer l_j mapped on PE_i. These sizes (in number of elements) are approximated based on the type of DNN layer l_j. If input data for layer l_j is simply the stored output from a previous layer on the same PE, we set the input buffer memory $M^j_{inbuffs}$ to zero. This prevents redundant memory calculations and optimizes overall memory usage. As an example, we give the formulas for the memory usage of three specific types of CNN layers (i.e., convolution, fully connected, and pooling).

Convolution layer:

$$M^j_{coeffs} = k^2 \times C_{in} \times C_{out} + C_{out}$$

$$M^j_{inbuffs} = H_{in} \times W_{in} \times C_{in}$$

$$M^j_{outbuffs} = H_{out} \times W_{out} \times C_{out}$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2P - k}{S} + 1 \right\rfloor \; ; \; W_{out} = \left\lfloor \frac{W_{in} + 2P - k}{S} + 1 \right\rfloor$$

where H_{in} and W_{in} are the input height and width, k is the kernel/filter size, C_{in} is the number of input channels, C_{out} is the number of output channels, S is the stride size, and P is the padding size around the borders of the input. H_{out} and W_{out} are the width and height of output tensors of layer l_j.

Fully connected layer:

$$M^j_{coeffs} = C_{in} \times C_{out} \; ; \; M^j_{inbuffs} = C_{in} \; ; \; M^j_{outbuffs} = C_{out}$$

Pooling layer:

$$M^j_{inbuffs} = H_{in} \times W_{in} \times C_{in}$$

$$M^j_{outbuffs} = H_{out} \times W_{out} \times C_{out}$$

$$H_{out} = \left\lfloor \frac{H_{in} - k}{S} + 1 \right\rfloor \; ; \; W_{out} = \left\lfloor \frac{W_{in} - k}{S} + 1 \right\rfloor$$

where H_{in} and W_{in} are the input height and width, H_{out} and W_{out} are the output height and width, k is the size of the pooling window, S is the stride of the pooling operation. $C_{in} = C_{out}$ is the number of input and output channels. Here, $M^j_{coeffs} = 0$, as pooling layers do not include coefficients.

Energy Consumption: Every PE_i consumes energy E^i to execute the DNN layers mapped on PE_i. In our energy consumption analytical model, E^i includes the energy consumed for inference computation and data communication with other PEs:

$$E^i = \sum_{\forall j : 1 \leqslant j \leqslant L \wedge x_j = PE_i} E^j_{comp} + \sum_{\forall j : 1 \leqslant j \leqslant L \wedge x_j = PE_i} E^j_{comm}$$

where E^j_{comp} and E^j_{comm} denote the computation and communication energy consumption for layer l_j, respectively. Here, E^j_{comm} is non-zero only when data produced or consumed by layer l_j actually has to be communicated with another PE. We calculate E^j_{comp} and E^j_{comm} as follows:

$$E^j_{comp} = \int_0^{t^j_{comp}} P^j_{comp}(t)\, dt; \quad E^j_{comm} = \int_0^{t^j_{comm}} P^j_{comm}(t)\, dt$$

where $P^j_{comp}(t)$ is the power consumption during the execution of layer l_j, and $P^j_{comm}(t)$ is the power consumption during data communication of layer l_j with another PE. $P^j_{comp}(t)$ and $P^j_{comm}(t)$ can be acquired by measurements during DNN layer profiling on an edge device.

Developing accurate analytical models to capture these objectives poses significant challenges. The complexity of creating these models stems from the necessity to precisely capture the intricate interactions and behaviors of various system components. Any misrepresentations or oversimplifications in these models can lead to substantial discrepancies between predicted and actual performance, thereby reducing the reliability of the estimation results.

2.3.3.2 *Real Measurement System-based Fitness Function*

Evaluating design solutions directly on physical systems offers the most accurate insights, as it mirrors real-world operational conditions without the distortions of analytical modeling. In our research, we focus on assessing various distribution configurations of DNN models at the Edge. To

facilitate this evaluation, we utilize the NVIDIA Jetson Xavier NX Developer Kit [27] as our primary testing platform in our thesis. Our evaluation system for distributed DNN inference includes eight Jetson NX devices interconnected over a network switch, forming an edge cluster. This setup allows us to mimic a distributed edge computing environment where multiple devices work collaboratively to handle diverse computational tasks. With this cluster, we can effectively test and validate our designs under realistic conditions that closely replicate the intended deployment scenarios for edge computing applications.

The NVIDIA Jetson Xavier NX device is exemplary for computational-demanding and AI tasks, particularly within embedded and edge systems. A simplified description of the device's technical specifications is presented in Table 2.1. It is equipped with an array of specialized hardware components such as 384 NVIDIA CUDA® Cores, 48 Tensor Cores, a 6-core Carmel ARM CPU, and two NVIDIA Deep Learning Accelerators (NVDLA) engines. These components collectively deliver up to 21 TOPS (Tera Operations Per Second), ensuring efficient processing of complex DNNs and high-resolution data from various sensors. The Jetson Xavier NX features 16 GBytes of LDDR4 DRAM as its main memory, accessible to all processors. It is crucial for storing the input data from sensors, and coefficients of DNN models for inference.

Table 2.1: Jetson Xavier NX 16GB Technical Specifications

AI Performance	21 TOPS (INT8)
GPU	384-core NVIDIA Volta™ GPU
CPU	6-core NVIDIA Carmel ARM®v8.2 CPU
Cache	6MB L2 + 4MB L3
Memory	16 GB 128-bit LPDDR4x
Storage	16 GB eMMC 5.1
Power	10W | 15W | 20W
Display	2x DP 1.4/eDP 1.4/HDMI 2.0
DL Accelerator	2x NVDLA
Vision Accelerator	2x PVA
Networking	10/100/1000 BASE-T Ethernet

Using a fitness function based on measurements taken from a real system offers a direct approach to evaluating the performance of distributed DNN inference in practical scenarios. However, this introduces significant challenges, primarily due to the extensive time required to conduct experiments and the engineering efforts to implement various designs. Physically setting up and executing each design solution involves a substantial amount of engineering work and time. Without adequate automation, relying on a real-system fitness function for evaluation becomes impractical due to these constraints.

In this thesis, we demonstrate that with adequate automation in place, it is feasible to use this approach effectively during Design Space Exploration (DSE). By automating the setup, execution, and measurement processes, we can streamline the evaluation of design solutions, thereby reducing the time and effort required. This approach not only makes it practical to employ real-system metrics in assessing fitness but also enhances the efficiency and accuracy of the DSE process, allowing for a more rapid search for potential solutions.

2.4 INTEROPERABILITY

Interoperability is a critical aspect of deploying Deep Neural Networks (DNNs) across various edge computing platforms. The challenge lies in the diverse landscape of deployment frameworks, deep learning toolsets, and the varying hardware capabilities of each device. This diversity can create significant barriers to seamless interaction and functionality across different platforms.

In environments from cloud servers to edge devices, and from CPUs to GPUs, each piece of hardware has its own set of characteristics and capabilities. This variation in computational resources across different edge devices underscores the critical need for interoperability, which can effectively bridge these disparities. Implementing interoperability allows for the flexible and efficient utilization of these diverse resources, particularly in the context of distributed DNN inference. Achieving interoperability involves the below two aspects:

Framework Compatibility ensures that DNN models can be deployed across different deep learning frameworks (like TensorFlow [28], PyTorch [29], etc.) without significant modifications. To address this challenge, developers can utilize tools like ONNX (Open Neural Network Exchange), which provides an open-source format for AI models. ONNX enables models to

be portable across different frameworks and hardware, thus promoting interoperability. Additionally, leveraging APIs that abstract away hardware-specific details can help developers focus on optimizing the DNN's performance without tying the model to one specific type of hardware. By converting models to ONNX, they can run directly on a variety of platforms and devices using ONNX Runtime [30]. This simplifies the deployment process, allowing DNN models to be executed seamlessly across different edge devices without the need for extensive optimization for each combination of framework and hardware. However, while ONNX addresses many interoperability challenges, ongoing efforts are needed to improve model conversion tools and ensure broader adoption of open standards to create a more integrated and flexible edge computing ecosystem. By integrating ONNX format support into our AutoDiCE framework, we tackle the interoperability issues related to the distributed deployment of DNN models, ensuring that DNNs can be effectively and efficiently integrated across diverse edge devices in practical deployment scenarios.

Standardized Communication between various components in distributed computing environments, such as those required for deploying deep neural networks across diverse platforms, can be effectively managed using the Message Passing Interface (MPI) [31]. The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. This protocol is commonly used for programming parallel computers, both in high-performance computing (HPC) and in the broader field of distributed computing. By using MPI, developers can ensure that data and model parameters are effectively communicated across different nodes in the network, supporting the synchronization required for training and inference in distributed DNNs. In our work, MPI provides a standardized communication protocol that enhances the interoperability and performance of DNN deployments at the Edge.

Part I

DISTRIBUTED DNN INFERENCE AT THE EDGE

The first part of the thesis focuses on the flexible and efficient DNN deployment across multiple edge devices. We delve into the innovative AutoDiCE framework in detail, which is designed to facilitate the flexible and fast distributed DNN deployment at the edge. Based on AutoDiCE, we propose an advanced two-stage Design Space Exploration (DSE) method, based on the NSGA-II algorithm introduces in Chapter 2, to explore efficient DNN mappings for DNN distribution.

Chapter 3 introduces AutoDiCE, an innovative, fully automated tool tailored for the distributed deployment of Convolutional Neural Network (CNN) models on multiple resource-constrained edge devices. This tool automates the process of splitting a CNN model into sub-models and facilitates the code generation needed for their distributed and collaborative execution across multiple, potentially heterogeneous, resource-constrained edge devices.

Chapter 4 introduces our advanced DSE methodology, incorporating a novel genetic encoding method to efficiently explore pipelined distributions of CNN models. The goal is to improve overall system inference throughput, reduce per-device energy and memory demands, and achieve optimal or near-optimal solutions that consider these three objectives, leveraging the AutoDiCE framework developed in Chapter 3.

3

CHAPTER 3

This chapter presents the proposed AutoDiCE framework in detail, for the automated splitting of a CNN model into a set of sub-models and automated code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices, while supporting the exploitation of parallelism among and within the edge devices. Our experimental results show that AutoDiCE can deliver distributed CNN inference with reduced energy consumption and memory usage per edge device and may in certain cases improve system throughput.

This Chapter is based on the journal article:

- **Xiaotian, Guo**, Andy D. Pimentel, and Todor Stefanov. *"Automated Exploration and Implementation of Distributed CNN Inference at the Edge"* [32], in *IEEE Internet of Things Journal*, 10.7 (2023): 5843-5858.

3.1 INTRODUCTION

Deep Learning (DL) [33] has become a popular method in AI-based applications in various fields, including computer vision, natural language processing, automotive, and many more. Especially, DL approaches based on convolutional neural networks (CNNs) [34] have been extensively utilized because of their huge success in image classification [35] and speech recognition applications [36].

Due to the high complexity of state-of-the-art CNN models, the training of these models is performed mainly on high-performance platforms, while the model inference is usually provided as a cloud service [37], allowing less powerful Internet-of-Things (IoT) devices at the Edge to easily use such services. Realizing CNN inference on edge devices using cloud services, however, requires users to communicate a substantial amount of data between an edge device and a cloud server. Such data communication may cause data privacy concerns as well as low device responsiveness due to data transmission delays or temporal unavailability of cloud services. Evidently, this is highly undesirable for those CNN-based applications that are particularly sensitive to compute response delays or the privacy of the processed data. For example, CNN-based navigation in self-driving cars [38] cannot tolerate variable and large response delays occurring due to the communication between the car and a cloud server. Or, applications in healthcare [39] using CNNs on IoT devices dealing with patient data cannot send their data to the cloud because this could lead to leakages of private data and violation of patients' privacy rights. The aforementioned concerns motivate the shift of the CNN inference from the Cloud to the Edge. When entirely executed at the Edge, a CNN is deployed close to the source of data, and data communication with a cloud server is not required, thereby ensuring high application responsiveness and reducing the risk of private data leakage.

Unfortunately, deploying and inferring a large CNN, which is typically memory/power-hungry and compute-intensive, on an IoT edge device is challenging because many edge devices have limited energy budgets and compute and memory resources. One approach to address this challenge is to construct a lightweight CNN model from a large CNN model by utilizing model compression techniques (e.g., pruning [40], quantization [41], knowledge distillation [42]), thereby reducing the CNN model size to a degree that allows the CNN to be deployed and efficiently executed on a resource-constrained edge device. However, the accuracy of the compressed CNN

model is significantly decreased if high compression rates are required. Another approach is to infer only part of a large CNN model on the edge device and the rest on the cloud by efficiently partitioning the model and distributing the partitions along the edge-cloud continuum [14]. However, the aforementioned edge device responsiveness and private data leakage issues are still inevitable in such partitioned CNN inference due to the partial involvement of the cloud. Finally, a third approach to address the challenge is to leverage all available resources along multiple, possibly heterogeneous, edge devices to deploy and execute a large CNN by properly partitioning the CNN model and running each CNN partition on a separate edge device. The size of each CNN partition should match the limited energy, memory, and compute resources of the edge device the partition runs on. Such an approach not only makes it possible to deploy large CNN models without the need of model compression, respectively without loss of accuracy, but it also resolves the aforementioned responsiveness and privacy issues because a cloud server is not involved in the CNN inference. Thus, in this chapter, we focus on this last approach, i.e., entirely distributing and executing a large CNN model at the Edge.

Although there are several approaches (e.g., [17–20]) that address distributed CNN inference on multiple edge devices, these efforts mostly focus on performance optimization through partitioning, scheduling, and exploiting parallelism. However, they typically do not address the actual partitioning itself as well as the deployment of the partitioned CNNs on the edge devices (i.e., collaborative inference), which still requires a significant manual design and programming effort. More specifically, it typically involves advanced skills in CNN model design, embedded systems programming, and parallel programming for (heterogeneous) distributed systems. At this moment, no design and programming framework exists that takes a trained ONNX model, together with a CNN partitioning specification, and **fully automates** the CNN model splitting and deployment on multiple edge devices to facilitate distributed CNN inference at the Edge. Therefore, in this chapter, we propose a framework, called **AutoDiCE**, for the automated splitting of a CNN model into a set of sub-models and automated code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices, while supporting the exploitation of parallelism *among* and *within* the edge devices. As such, our framework is complementary to the aforementioned, existing approaches because it targets the actual automation of splitting, code generation, and model deployment for distributed CNN inference at the Edge.

To the best of our knowledge, this is the first framework that offers the following features:

- A tool, called AutoDiCE, featuring the automated splitting of a CNN model into a set of sub-models and automated code generation for distributed and collaborative execution of these sub-models on multiple, possibly heterogeneous, edge devices. AutoDiCE is the first fully automated tool for distributed CNN inference over multiple resource-constrained devices at the Edge. It is open-source and available at [43];
- A hybrid MPI and OpenMP code generation approach in AutoDiCE to support the exploitation of parallelism *among* and *within* the edge devices, i.e., the latter exploiting multi-core execution;
- A highly flexible AutoDiCE implementation that facilitates easy specification and reuse of existing CNNs (via the ONNX format [44]), and can target a range of (heterogeneous) edge devices via a custom inference engine library which supports a variety of CPUs (x86, ARM), GPUs (NVIDIA, Mali, AMDRX) and GPU APIs (VULKAN, CUDA);
- A range of experiments in which we show that our framework AutoDiCE can rapidly realize a wide variety of distributed CNN inference implementations on multiple edge devices, achieving improved (i.e., reduced) per-device energy consumption and per-device memory usage, and under certain conditions, improved system (inference) throughput as well.

The remainder of the chapter is organized as follows. Section 3.2 discusses related work, after which Section 3.3 presents our AutoDiCE tool. In Section 3.4, we describe a range of experiments, demonstrating that our framework can rapidly explore and realize a wide variety of distributed CNN inference implementations with diverse trade-offs regarding energy consumption, memory usage, and system throughput. Section 3.5 provides a discussion on the current version of our framework and how it could be further improved in the future. Moreover, we further clarify, with examples, why distributed CNN inference using our novel framework is beneficial in real-world application scenarios when the CNN memory footprint and energy consumption are a concern. Finally, Section 3.6 concludes the chapter.

3.2 RELATED WORK

Today's convolutional neural network (CNN) models for computer vision tasks are becoming increasingly complex. For example, the CNN-based model CoAtNet-7 [45] reaching Top-1 accuracy of 90.88% for the ImageNet dataset has 2.44 billion parameters (weights and biases) which values have to be determined during the training and stored/used during the inference. To train and deploy such large CNN models, parallel or distributed computing is often required. For model training, a common approach to accelerate the training process is to exploit pipeline parallelism. For example, GPipe [46] applies pipeline parallelism by splitting a mini-batch of training data into smaller micro-batches, where different GPUs train on different micro-batches. Another example is PipeDream [47] which partitions the CNN model for multiple GPUs such that each GPU trains a different part of the model. An alternative distributed training approach, motivated by privacy concerns among multiple devices/machines, is federated learning (FL) [48, 49]. FL aims at training a global centralized model with multiple, local datasets on distributed devices or data centers, thereby preserving local data privacy and improving learning efficiency. All of the aforementioned approaches target efficient, distributed training of large CNN models. In contrast, our work presented in this chapter focuses on efficient, distributed inference of large CNNs.

Unlike CNN training, the inference of large CNN models often needs to take multiple requirements into account, such as latency, throughput, resource usage, power/energy consumption, etc. To satisfy these requirements for CNN inference on edge devices, CNN models are typically distributed along *the cloud-edge continuum*, or *fully at the Edge*.

CNN inference along *the cloud-edge continuum* (e.g., [14–16]) deploys the CNN computations on the cloud and the Edge. Such an approach maximizes the utilization of computing resources on edge devices, reduces the computation workload on the cloud, and usually improves the CNN inference throughput. The most common idea in this approach is to obtain a specific small sub-model from or an early-exit branch of the initial large CNN model that runs on the edge device. Only if the inference result of the deployed sub-model/early-exit branch on the edge device is below a certain confidence threshold, the device has to upload its data to the cloud and the CNN inference has to continue on the cloud. Vertical distribution along the cloud-edge continuum still relies on the quality and stability of network connections between the edge device and the cloud server. This

not only suffers from high communication latency but also there is a risk of information leakage. In contrast, our framework achieves lower inference latency by deploying a large CNN model over edge devices without the cloud, and therefore also preserves both data and model privacy.

CNN inference (e.g., [17–20]) *fully at the Edge* distributes the workload of a large CNN across multiple edge devices without the cloud. That is, all CNN computations are collaboratively executed at the Edge and there is no dependency on the cloud. Data partitioning and model partitioning are two common methods. Model partitioning includes horizontal partitioning and vertical partitioning which are discussed in detail in Chapter 2.. Data partitioning exploits data parallelism among multiple devices by splitting the input/output data to/from CNN layers into several parts while each device executes all layers of a CNN model using only some parts of the data. For example, DeepThings [18] uses the Fused Tile Partitioning (FTP) method for splitting input data frames of CNN layers in a grid fashion to reduce the CNN memory usage. The main drawback of the data partitioning method is that an edge device should still be capable of executing all layers of a CNN model which implies that the Edge device should be able to store the weights and biases of the entire CNN model. Alternatively, the model partitioning method splits the CNN layers and/or connections of a large CNN model horizontally or vertically, thereby creating several smaller sub-models (model partitions) where each sub-model is executed on a different edge device [19]. For example, MoDNN [17] splits convolution layers and fully connected layers in the VGG-16 model. In [20], CNN layer connections are split and each CNN layer is treated as a sub-task. These sub-tasks are then mapped to edge devices through a balanced processing pipeline approach.

The model partitioning methods for distributed inference focus on performance optimization through partitioning, scheduling, and exploiting parallelism. Complementary to these efforts, our work focuses on the actual automation of the splitting of and code generation for a CNN model, together with the actual model deployment on distributed devices at the Edge. As will be demonstrated in Chapter 4, this facilitates, e.g., very accurate design space exploration (DSE). That is, our framework is designed to be flexible enough for users to accurately and easily explore different design objectives of distributed CNN inference at the Edge such as reducing memory usage and energy consumption per edge device in order to find an efficient distribution and deployment of a CNN model.

3.3 THE AUTODICE TOOL

In this section, we present our AutoDiCE tool, which is designed for the efficient and flexible deployment of distributed CNN inference implementations at the Edge. To this end, we describe AutoDiCE as a design flow and explain the main steps in the flow with the help of an illustrative example. First, we provide a high-level overview of the AutoDiCE design flow. Second, we describe AutoDiCE's unified user interface. Next, we explain in detail the main steps in the front-end of the AutoDiCE design flow. Finally, we do the same for the back-end of the flow.

3.3.1 *Overview*

AutoDiCE is a flexible tool that facilitates distributed inference of a CNN model, embedded in an AI application, at the Edge. More specifically, it allows designers and programmers of such CNN-based AI applications to perform, *in a fully automated manner*, CNN model partitioning, deployment, and execution on multiple resource-constrained edge devices. Figure 3.1 shows the AutoDiCE user interface and design flow where the main steps in the flow are divided into two modules: front-end and back-end.

The interface is composed of three specifications, namely Pre-trained CNN Model provided as an .onnx file, Mapping Specification provided as a .json file, and Platform Specification provided as a .txt file.

The Pre-trained CNN Model specification includes the CNN topology description with all layers and connections among layers as well as the weights/biases that are associated with the layers and obtained by training on a specific dataset using deep learning frameworks like PyTorch, TensorFlow, etc. Many such CNN model specifications in ONNX format [44] are readily available in open-access libraries and can be directly used as an input to AutoDiCE.

The Platform Specification lists all available edge devices together with their computational hardware resources and specific software libraries associated with these resources. This specification is simple to draw up and can be generated by external tools that query the network connecting the edge devices or provided manually by the user.

The Mapping Specification is a simple list of key-value pairs in JSON format that explicitly shows how all layers described in the Pre-trained CNN Model specification are mapped onto the computational hardware resources listed in the Platform Specification. Every unique key corresponds

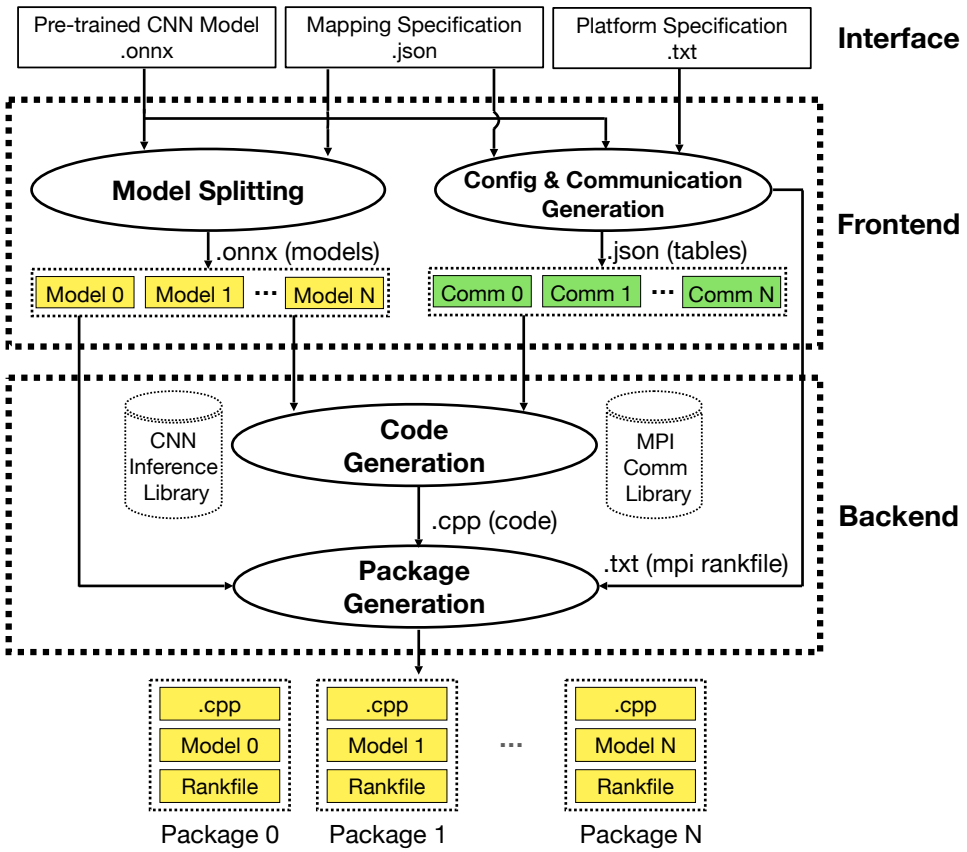


Figure 3.1: The AutoDiCE design flow and its user interface

to an edge device with a selection of its hardware resources to be used for computation. Every value corresponds to a set of CNN layers to be deployed and executed on the edge device resources. Such a Mapping Specification can be provided manually by the user or, like will be discussed in Chapter 4, generated by a system-level design-space exploration (DSE) tool.

The three aforementioned specifications are given as an input to the front-end module as shown in Figure 3.1. Two main steps are performed in this module: *Model Splitting* and *Config & Communication Generation*. The Model Splitting takes as an input the Pre-trained CNN Model and Mapping specifications, splits the input CNN model into multiple sub-models, and generates these sub-models in ONNX format. The number of generated sub-models is equal to the number of unique key-value pairs in the Mapping Specification. Each sub-model contains input buffers, output buffers,

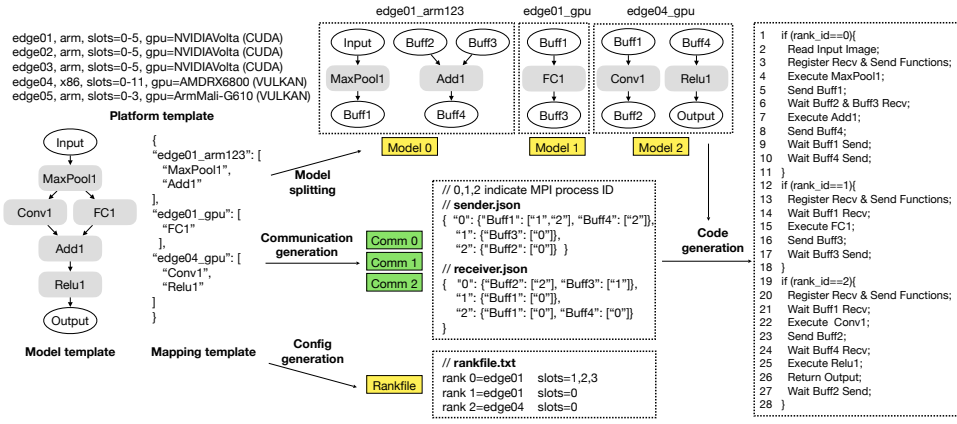


Figure 3.2: AutoDiCE in action: a detailed example

and the set of CNN layers, specified in the corresponding key-value pair. The Config & Communication Generation step takes all three specification files as an input and generates specific tables in JSON format containing information needed to realize proper communication and synchronization among the sub-models using the well-known MPI interface. In addition, a configuration text file (MPI rankfile) is generated to initialize and run the sub-models as different MPI processes.

As shown in Figure 3.1, the generated configuration file, sub-models, and tables are used in the back-end module for code and deployment package generation. During the *Code Generation* step in this module, efficient C++ code is generated for every edge device based on the input sub-models and tables. In the generated code, primitives from the standard MPI library are used for data communication and synchronization among sub-models as well as primitives from our customized CNN Inference Library are used for implementation of the CNN layers belonging to every sub-model. Both libraries enable the generation of cross-platform code that can be compiled for and executed on multiple heterogeneous edge devices. Finally, the *Package Generation* step packs the generated cross-platform C++ code, the MPI rankfile, and a sub-model together to generate a specific deployment package for every edge device. All packages contain the same C++ code and the same MPI rankfile but different sub-models. When a package is compiled, deployed, and executed on an edge device, the specific sub-model in the package will be loaded and only the part of the code that corresponds to the loaded sub-model will run as an MPI process as specified in the MPI configuration rankfile.

In the following subsections, the interface and the main steps of the AutoDiCE design flow, introduced above, are explained in more detail with the example in Figure 3.2.

3.3.2 *Interface*

In the left-most part of Figure 3.2, we show three templates (examples) representing the three specifications of the user interface introduced in Section 3.3.1. By using these example templates, we comprehensively reveal and explain the flexibility of and heterogeneity support in AutoDiCE.

In general, the Platform Specification lists all available edge devices with their computational resources. Every line in the list specifies the name of the edge device, the CPU architecture, the number of CPU cores, and (optionally) a GPU device with its architecture and programming library. For instance, the first line of the Platform template in Figure 3.2 specifies that the name of the device is *"edge01"* with an ARM processor architecture including six cores in total (slots=0-5) and one GPU device with NVIDIA Volta architecture supported by the CUDA library. Through the Platform Specification, a user can easily and flexibly specify alternative heterogeneous hardware platforms including different numbers of edge devices and types of resources. As shown in Figure 3.2, the user can select different CPU architectures per edge device such as ARM, x86, etc. with different numbers of cores as well as different GPU architectures per edge device such as NVIDIA, Mali, AMDRX, etc. with different GPU programming APIs such as CUDA, VULKAN, etc.

The Model template in Figure 3.2 is an example of a part of a Pre-trained CNN Model specification that visualizes the CNN model topology only. It contains an input layer, five hidden layers (i.e., MaxPool1, Conv1, FC1, Add1, and Relu1), and an output layer. Every hidden layer stores its own parameters (such as weights, bias, etc.) that are not shown in Figure 3.2. We adopt ONNX as the standard format to represent/specify a pre-trained CNN model in the AutoDiCE interface for framework interoperability. The choice of ONNX allows users to provide a CNN model designed, trained, and verified in well-known and widely-used frameworks such as TensorFlow [28], PyTorch [29], etc. A large variety of trained CNN models are already available in ONNX format that can be readily utilized by AutoDiCE, allowing easy deployment of these models over multiple edge devices. In addition, the use of the ONNX interface facilitates reproducibility in terms of CNN designs (e.g., CNN topology, used parameters,

etc.) and in CNN evaluations (for CNN model accuracy and non-functional characteristics). For example, in experimental evaluations, users can confidently and reliably compare CNN model characteristics such as accuracy, memory usage, performance, and power/energy consumption, obtained by AutoDiCE, with the same characteristics obtained by other frameworks and approaches, applied to exactly the same CNNs.

As mentioned in Section 3.3.1, the Mapping specification lists several different key-value pairs to describe a distribution of the layers in a CNN model over different computational platform resources. The Mapping template in Figure 3.2 is an example of such specification. It lists three different key-value pairs. For example, the unique key *"edge01_arm123"* specifies that three ARM CPU cores (i.e., cores 1, 2, and 3) of device *edge01*, described in the Platform specification, are allocated for CNN layers execution. The corresponding value [*"MaxPool1", "Add1"*] specifies that layers MaxPool1 and Add1, described in the Pre-trained CNN model specification, are executed on the allocated three cores. All valid keys must be generated from the Platform Specification to ensure the availability of chosen computational resources. CNN layers can be bound to a single GPU, a single CPU core, or multiple CPU cores. Specifically, if all keys use computational resources of the same device, the distributed inference turns into a multi-threaded execution on a single device. All valid values must be selected from layers of the Pre-trained CNN model, and all CNN layers in that model should be assigned to at least one hardware processing unit (CPU or GPU) to ensure the mapping consistency. The mapping example in Figure 3.2 is a vertical partitioning (Figure 2.3), which means that every CNN layer is mapped to a single unique key (device). If a CNN layer is mapped to multiple unique keys, then the layer will be horizontally distributed over multiple computational resources. Users can realize different approaches for splitting (and parallel execution of) a CNN model, namely vertical, horizontal, and using data parallelism (the latter two are not shown in Figure 3.2). This is done by changing the layer distribution in the Mapping Specification. It is easy and flexible for users (or DSE and other tools, for that matter) to change the CNN model partitioning as well as the mapping of partitions to edge devices through selecting different combinations of key-value pairs in the Mapping Specification.

3.3.3 *Front-end*

The front-end module is designed to parse, check, and pre-process all user specifications through its two main steps: Model Splitting and Config & Communication Generation. Model Splitting splits the input CNN model according to the mapping specification and generates several CNN sub-models. Each sub-model will be implemented and executed as an MPI process. Config & Communication Generation generates an MPI-specific configuration file and communication tables based on the three input specification files. At the top center of Figure 3.2, the model splitting step is illustrated. Based on the three key-value pairs in the Mapping template (specification), the CNN model template is vertically partitioned into three sub-models (*Model 0, Model 1, and Model 2*). The layers of the CNN model mapped on the same edge device resource will be grouped into a single sub-model. For example, the two layers *MaxPool1* and *Add1* are grouped together to form sub-model *Model 0*.

The output of a CNN layer in the initial Model template is the input of its next connected CNN layers. If two connected CNN layers are mapped onto different edge devices or different compute resources (CPU or GPU) within an edge device, i.e., the two layers belong to two different sub-models, the direct connection between these two layers is replaced by one output buffer belonging to one of the sub-models and one input buffer belonging to the other sub-model. These two buffers are used to store and communicate intermediate results between the two CNN layers. For example, the directly connected CNN layers *MaxPool1* and *Conv1* of the Model template in Figure 3.2 are mapped onto two different edge devices according to the Mapping template. Thus, layer *MaxPool1* belongs to sub-model *Model 0* and layer *Conv1* belongs to sub-model *Model 2*. As a consequence, the direct connection between *MaxPool1* and *Conv1* is replaced by output buffer *Buff1* in *Model 0* and input buffer *Buff1* in *Model 2*.

The Config Generation step is illustrated in the bottom center of Figure 3.2. It generates an MPI-specific Rankfile which provides detailed information about how the individual MPI processes, corresponding to the generated sub-models, should be mapped onto edge devices, and to which processor/core(s) of an edge device an MPI process should be bound. In the example in Figure 3.2, we have three sub-models *Model 0, Model 1, and Model 2* that will be implemented and executed as three different MPI processes 0, 1, and 2, respectively. Based on the Mapping template, the example Rankfile in Figure 3.2 specifies that the MPI processes 0 and 1 should be

mapped onto edge device *edge01* and the MPI process 2 should be mapped onto edge device *edge04*. In addition, each line of the Rankfile specifies the physical processors/cores allocated to the corresponding MPI process. In our example Rankfile, the first line specifies that MPI process 0 should be mapped on edge device *edge01* and slots 1, 2, and 3 are allocated to this process on this device. This means that this process will run on three ARM CPU cores (i.e., core 1, 2, and 3) of device *edge01*.

The Communication Generation step is illustrated in the center of Figure 3.2. It generates a sender table and a receiver table as .json files. These two communication tables specify the necessary communications between individual MPI processes to ensure that the input/output buffers of the corresponding sub-models are synchronized through the MPI interface. For example, the first line in the sender table specifies that MPI process 0 needs to send the contents of *Buff1* to MPI processes 1 and 2, and the contents of *Buff4* to MPI process 2. Correspondingly, the third line in the receiver table specifies that MPI process 2 needs to receive the contents of *Buff1* and *Buff4*, both from MPI process 0. The communication and synchronization information in the sender and receiver tables ensure that the initial input CNN model is correctly executed after the model splitting.

3.3.4 *Back-end*

The back-end module constitutes AutoDiCE's final stage to create a CNN-based application for deployment over multiple edge devices. It contains two main steps: Code Generation and Package Generation.

The first step, Code Generation, turns all intermediately generated files (all sub-models and communication tables) by the front-end module into efficient C++ code. The output of this step is a single .cpp file which has a very specific and well-defined code structure, making calls to specific primitives and functions located in two libraries: a standard MPI Library and our customized CNN Inference Library. The code structure contains several code blocks. Each code block is surrounded by an *if* statement and implements one CNN sub-model. The sub-models are executed as individual MPI processes mapped on different edge device resources, meaning that every MPI process runs only the code block implementing the corresponding sub-model. The code block is uniquely identified by a rank ID checked in the *if* statements surrounding the code blocks. Unique rank IDs are assigned according to the Rankfile, explained in Section 3.3.3, during the MPI initialization stage. The pseudo-code template in the right-most part of

Figure 3.2 illustrates the specific code structure of the generated .cpp file. It contains three code blocks, i.e., Lines 1-11, Lines 12-18, and Lines 19-28, that implement sub-models *Model 0, Model 1, and Model 2*, respectively. *Model 0, Model 1, and Model 2* will be executed as three MPI processes 0, 1, and 2, respectively. Every MPI process contains the aforementioned code template but the MPI process 0 corresponding to sub-model *Model 0* will run only the code block between lines 1 and 11. Similarly, the MPI process 1 will run only the code block between lines 12 and 18, etc.

The code blocks themselves all have a similar, well-defined structure starting with code that registers all MPI send and receive primitives (e.g., lines 3, 13, and 20 in Figure 3.2) followed by MPI_Wait primitives that block the code execution until the necessary data to be processed by CNN layers is received (e.g., lines 6, 14, 21, and 24). Then, code implementing the CNN layers is executed followed by MPI_Send primitives that communicate the output data from a layer to other layers executing in different MPI processes mapped on different edge devices/resources (e.g., lines 7-8, 15-16, 22-23). Finally, MPI_Wait primitives are used to block the code execution until the sent data arrives at the destination (e.g., lines 9, 10, 17, and 27).

Some code blocks have to implement and execute more than one CNN layer because the corresponding CNN sub-models contain multiple CNN layers. Every code block implementing multiple CNN layers has to execute the layers in the order specified by the data dependencies in the input CNN Model template to preserve the functional correctness of the distributed CNN model. For example, the CNN sub-model *Model 0* in Figure 3.2 is implemented by the code block between lines 1 and 11 in Figure 3.2. Line 2 reads an image file to prepare the input data for the CNN model. The code in line 3 registers all non-blocking MPI send and receive primitive calls according to the first lines in the sender and receiver tables, explained in Section 3.3.3. In lines 4 and 7, the *MaxPool1* and *Add1* layers are executed one after the other, thereby preserving the order specified in the CNN Model template given in Figure 3.2. After executing each layer, they store their output data in *Buff1* and *Buff4*, respectively. Line 5 sends the content of *Buff1* to MPI process 1 and MPI process 2 according to the sender table. To allow for overlapping communication with computation, the generated code uses non-blocking MPI_Send primitives that return immediately and will not block the execution. A layer within a code block is executed once its input data is available, i.e., layers are executed in a data-driven fashion. For those layers that read their input data from communication buffers (i.e., data generated by another sub-model, possibly running on a different

edge device), MPI synchronization (wait) primitives enforce that layers cannot start execution before their input data is available. For example, this data-driven based execution of layers enforces that the *Add1* layer in *Model 0* can only be executed after the input data in *Buff2* and *Buff3* is available. Such synchronization is realized by the MPI_Wait primitives in line 6 of Figure 3.2. Line 8 uses the non-blocking MPI_Send primitive again to transfer the content of *Buff4* to MPI process 2. Finally, at the end of the code block, in lines 9-10, two synchronization MPI_Wait primitives are called that are associated with the two asynchronous send requests in lines 5 and 8. All such synchronization primitives are always called at the end of a code block in order to stop the code execution until the corresponding send requests (in this example the requests to send the contents of *Buff1* and *Buff4*) are completed.

In every code block, the implementation and execution of the CNN layers is realized by calling functions and primitives located in our customized CNN Inference Library. By encapsulating the NCNN [50] and Darknet [51] neural network engines into a uniform wrapper, our custom inference library supports CNN layer implementation and execution on a variety of hardware platforms (e.g., Raspberry Pi with a quad-core ARM v8 SoC, NVIDIA Jetson AGX Xavier series, etc.).

The used MPI primitives in the code blocks are part of the Open MPI library [31], which is an open-source implementation of the standard MPI interface for high performance message passing. It enables parallel execution on both homogeneous and heterogeneous platforms without drastic modifications to the device-specific code.

Besides facilitating the C++ code generation and distributed execution of CNN models (using MPI), our customized CNN Inference Library also integrates and provides OpenMP support. This means that if a CNN layer is mapped onto multiple CPU cores in an edge device, the actual execution of such layer will be multi-threaded using OpenMP in order to efficiently utilize the multiple CPU cores by exploiting data parallelism available within the layer. For example, the *MaxPool1* layer in Figure 3.2 is implemented and executed as multiple threads within MPI process 0 which is mapped onto the three ARM CPU cores 1, 2 and 3 in edge device *edge01*. More specifically, in Figure 3.3, we show some details about how the multiple threads bound to the three CPU cores 1, 2 and 3 are executed within MPI process 0. A thread number variable, called *num_threads*, is set to 3 in the code block implementing MPI process 0 during the code generation step. In our customized CNN Inference Library, this variable is used in the implementation

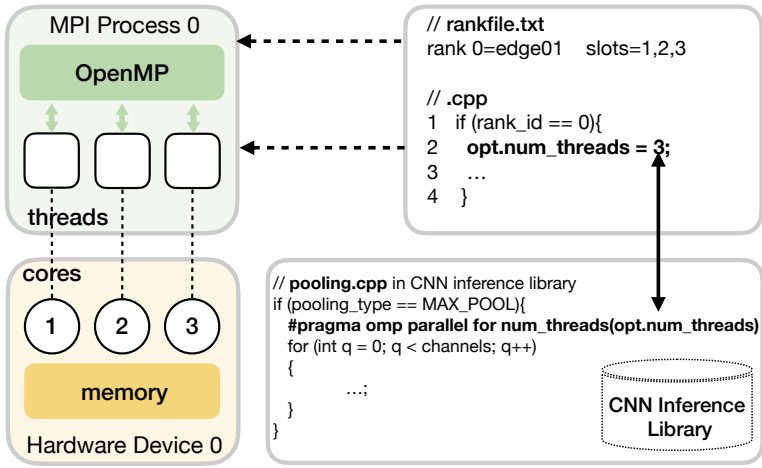


Figure 3.3: MPI process 0 with OpenMP

code of all types of layers (i.e., convolution, pooling, etc.), and it configures the OpenMP macro line *#pragma omp parallel for* shown in Figure 3.3. This macro line spawns a group of multiple threads and divides the loop iterations (the *for* loop in Figure 3.3) that follow this macro line between the spawned threads during the execution. So, during the execution, layer *MaxPool1* is executed as three threads running on CPU cores 1, 2, and 3.

The above discussion on the first step (Code Generation) of the back-end module clearly indicates that AutoDiCE employs a hybrid MPI+OpenMP programming model. OpenMP is used for parallel execution of a CNN layer within an edge device and MPI is used for communication and synchronization among CNN sub-models running on different edge devices or on different compute resources (e.g., CPUs and GPUs) within an edge device. By doing so, AutoDiCE provides extreme flexibility in terms of many alternative ways to distribute the CNN inference within and across edge devices by treating every CPU core or GPU unit in edge devices as a separate entity with its own address space. This allows AutoDiCE to be used in very complex IoT scenarios that may contain a lot of heterogeneous devices.

The second step of the back-end module, i.e. Package Generation, packs the generated .cpp code, sub-models, and Rankfile together into a deployment package for every edge device utilized in the distributed CNN inference. As it is essential to identify the individual MPI process running on an edge device, this step must put the Rankfile in every package. The Rankfile provides detailed information about the MPI processes' binding, which constrains each MPI process to run on specific compute resources of

different edge devices. The executable binary (to be deployed on an edge device) will be generated when the corresponding .cpp code in a package is compiled together with the aforementioned customized CNN Inference Library we have developed. As all packages contain the same .cpp code (i.e., we use the Single Program Multiple Data paradigm in this sense), the same binary can be deployed and executed on the same type of edge devices where each edge device will load the corresponding CNN sub-model from its own package before the execution of the binary. For different types of edge devices, we can generate an executable binary for every type.

3.4 FRAMEWORK EVALUATION

In this section, we present an evaluation of our AutoDiCE framework. First, we describe the setup for our experiments in Section 3.4.1. Then, in Section 3.4.2, we evaluate the execution time of our framework to show its efficiency. Moreover, we also present a range of experimental results for three representative CNNs to demonstrate that our novel framework can rapidly realize a wide variety of distributed CNN inference implementations with diverse trade-offs regarding energy consumption per device, memory usage per device, and overall system throughput. Finally, in Section 3.4.3, we analyze the effects on the energy consumption per device, the memory usage per device and the overall system throughput when scaling the distributed CNN inference to a varying number of deployed edge devices.

3.4.1 *Experimental Setup*

The goal of our experiments is to demonstrate that, thanks to our novel contributions presented in this chapter, our framework can rapidly explore and automatically implement CNN partitions over multiple edge devices to realize distributed CNN inference. Moreover, it can do so with lower per-device energy consumption, with smaller per-device memory usage, and under certain conditions, with the same or higher CNN inference throughput, as compared to CNN execution on a single edge device. Since state-of-the-art CNNs have deep architectures with many layers, this leads to an immense variety of different CNN mappings on multiple edge devices, each having different characteristics in terms of energy consumption per device, CNN inference throughput, and memory usage per device. Therefore, we have designed a design-space exploration (DSE) experiment (details in Chapter 4) to find Pareto-optimal CNN mappings with respect to

CNN inference throughput, energy consumption per device, and memory usage per device.

In the DSE experiment in this chapter, we use three real-world CNNs, namely VGG-19 [52], Resnet-101 [53], and Densenet-121 [54], from the ONNX models zoo [55] that take images as input for CNN inference. Since these CNNs are diverse in terms of types and number of layers, and memory requirements to store parameters (weights and biases), we believe that they are representative and good targets for our evaluations to demonstrate the merits of our framework. The first four columns in Table 3.1 list the details of the used CNN models.

The aforementioned CNN models are mapped and executed on our edge cluster introduced in Section 2.3.3.2. For a given CNN mapping specification, we apply our AutoDiCE framework to generate and distribute a deployment package for every Jetson device. For every generated implementation, we measure and collect the energy consumption per device, CNN inference throughput, and memory usage per device, as an average value over 20 CNN inference executions. As we target embedded devices, the batch size of CNN inference is 1. The inference throughput (measured by instrumenting the code with appropriate timers) and the memory usage per device are reported directly by the code itself during the CNN execution. To measure the energy consumption per device, a special sampling program reads power values from the integrated power monitors on each NVIDIA Jetson board during the CNN execution period, where the power consumption involves the whole board including CPUs, GPU, SoC, etc.

As discussed in Chapter 2, to actually explore the different CNN mappings, while optimizing for the three target objectives (i.e., system throughput, energy consumption per device, and memory usage per device), we apply the well-known NSGA-II [5]. The chromosomes in our NSGA-II multi-objective GA implementation encode how a CNN is split into different segments and how these segments are mapped onto the various edge devices and resources within them. To evaluate the fitness of the encoded CNN mappings using our AutoDiCE framework, the chromosomes are translated to the framework's mapping format described in Section 3.3.2. In our DSE experiment, every CNN layer can be mapped either onto a single CPU core, onto six CPU cores, or onto a GPU inside an edge device. The GA is executed with a population size of 100 individuals, a mutation probability of 0.1, a crossover probability of 0.5, and performs 400 search generations. For all experiments with the three CNNs, the original data precision (i.e., float32) is utilized to preserve the original model accuracy of classification.

Table 3.1: Used CNN models and AutoDiCE execution time breakdown

Network	Total # Layers	Total # Parameters	Memory for Parameters (MB)	AutoDiCE Execution Time (seconds)		
				Front-end	Back-end	Package deployment
DenseNet-121 [54]	910	8.06 million	32	1.93	0.3	21.3
ResNet-101 [56]	344	44.6 million	171	7.30	0.1	23.3
VGG-19 [52]	47	143 million	549	21.50	0.4	26.9

3.4.2 *Efficiency of AutoDiCE and DSE Results*

We start with evaluating the execution time of AutoDiCE itself, to provide insight on how long this tool generally takes to split a CNN model (front-end), to generate the code for the distributed CNN execution (back-end), and to deploy the generated packages to the edge devices for actual execution. To this end, we have measured the required time for each of these phases using the 'worst-case scenario' in the scope of our experiments: using the maximum number of splits in our CNNs to generate sub-models (24 splits/sub-models of a CNN in our experiments), and mapping and deploying the generated sub-models to the maximum number of edge devices (8 in our experiments). These measurements were done on a system equipped with an Intel Core i7-9850H processor, running Ubuntu 20.04.3 LTS. The last three columns in Table 3.1 provide a breakdown of the execution time (in seconds) of AutoDiCE for the three CNNs in these worst-case scenarios. From the results in Table 3.1, we can see that AutoDiCE is able to produce executable, distributed CNNs and deploy them on the various edge devices in a relatively short time frame, i.e., in less than a minute for any of the three used CNNs in our worst-case scenario. The comparatively larger execution time of the front-end for VGG-19 is due to the high number of parameters in this model, and the resulting overheads in AutoDiCE of copying these parameters to the large number of sub-models. In any case, these results demonstrate that AutoDiCE allows for rapidly splitting CNNs and deploying them for distributed execution on multiple edge devices.

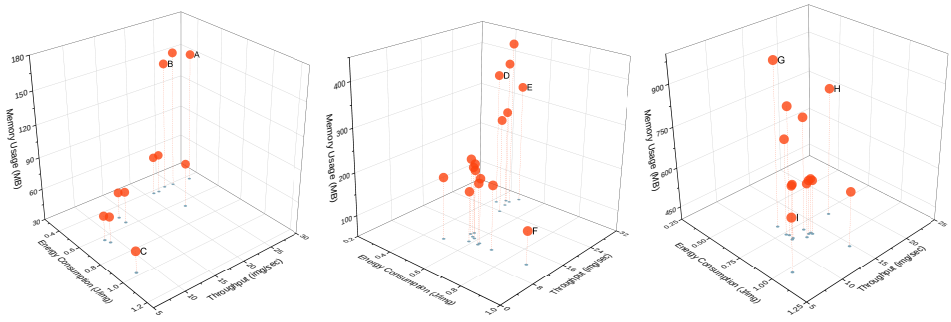
Our DSE experiments explore a wide range of different CNN mappings and these experiments result in a Pareto front with several Pareto-optimal mappings. In such a set of Pareto-optimal mappings, none of the targeted objectives (energy consumption, throughput, and memory usage) can be further improved without worsening some of the other objectives. More specifically, we consider the *maximum* energy consumption *per device*, *maximum* memory usage *per device*, and total system (CNN inference) through-

put as our target objectives. Figures 3.4a, 3.4b, and 3.4c show the Pareto-optimal CNN mappings found by our DSE for DenseNet-121, ResNet-101, and VGG-19, respectively. To better illustrate (the diversity of) these Pareto-optimal mappings, Table 3.2 shows more details about a selection of these mappings (points A to I in Figure 3.4) for comparison. As a reference, the table also includes the mapping results when using a single edge device with 6 CPUs or 1 GPU.

Moreover, to provide a feeling of how the distributed CNN execution on resource-constrained edge devices compares to CNN execution on a (centralized) powerful server, Table 3.2 also includes throughput and GPU memory results from an experiment on an NVIDIA GeForce RTX2080 Ti card (4352 NVIDIA CUDA cores and 544 Tensor cores, with a theoretical maximum performance of 13.45 TFLOPS) with Pytorch to mimic a cloud server based execution of the CNNs. Here, we would like to stress that the mimicked cloud server results do not include any latencies required for sending data to and from the cloud server, which would be the case in reality. To make a fair comparison with our experimental edge devices, the inference batch size when using the aforementioned NVIDIA GPU card is also set to 1. We note that it is not possible to precisely measure the energy consumption of the GPU card, thus its energy consumption is not given in Table 3.2. However, its energy consumption is definitely much higher compared to our experimental edge devices. For memory usage, we have taken the peak memory usage of the GPU card because it is influenced by the CNN model and its execution.

Columns 3 and 5 in Table 3.2 show the maximum energy consumption per device (in Joules per image) and maximum memory usage per device (in MegaBytes) for a specific CNN mapping, respectively. Column 4 shows the overall system throughput (in images per second). Columns 6, 7 and 8 show the hardware configurations of the selected CNN mappings, consisting of the number of deployed edge devices, and total of CPU cores and GPUs used in these devices, respectively.

From Figure 3.4 and Table 3.2, we can see that our novel framework allows for easily and rapidly realizing a wide variety of distributed CNN inference implementations with diverse trade-offs regarding per-device energy consumption, per-device memory usage, and overall system throughput. Taking point A as an example, a distributed execution of DenseNet-121 on four devices utilizing only GPUs can reduce the maximum energy consumption per device by 52.5% and 33.8% as compared to the 1-Device CPU and 1-Device GPU hardware configurations, respectively. The sys-



(a) DenseNet-121 (910 layers) (b) ResNet-101 (344 layers) (c) VGG-19 (47 layers)

Figure 3.4: Pareto-optimal CNN mappings from our DSE experiment with three CNNs.

Table 3.2: Selected Pareto-optimal Mappings (points) from Figure 3.4

Network	Points	Max. per-device Energy (J/img)	System FPS (img/sec)	Max. per-device Memory (MB)	# Devices	# CPU cores	# GPUs
	NVIDIA GTX 2080ti	-	21.339	286.854	-	-	1
	1-Device CPU	0.905	7.987	129.984	1	6	0
	1-Device GPU	0.650	12.807	251.172	1	0	1
DenseNet-121	A	0.430	**27.941**	152.336	4	0	4
	B	**0.408**	23.551	149.941	6	6	5
	C	0.977	7.546	**51.066**	8	38	0
	NVIDIA GTX 2080ti	-	30.823	437.446	-	-	1
	1-Device CPU	1.635	5.786	656.527	1	6	0
	1-Device GPU	1.031	21.767	955.012	1	0	1
ResNet-101	D	**0.425**	26.406	360.766	7	0	7
	E	0.488	**30.048**	329.641	7	12	5
	F	0.886	12.123	**127.883**	8	48	0
	NVIDIA GTX 2080ti	-	166.820	822.902	-	-	1
	1-Device CPU	1.471	7.273	1310.91	1	6	0
	1-Device GPU	1.523	11.664	1666.418	1	0	1
VGG-19	G	**0.680**	11.651	998.273	6	0	6
	H	0.791	**17.385**	868.496	6	6	5
	I	1.035	7.194	**604.504**	7	30	2

tem throughput of DenseNet-121 on four devices achieves a 3.5x and 2.2x performance improvement compared to the 1-Device CPU and 1-Device GPU configurations, respectively. In terms of per-device memory usage, the CNN mapping A with four devices consumes 39.3% less memory than the 1-Device GPU implementation, but consumes 17.2% more memory as compared to the 1-Device CPU configuration. Moreover, the distributed CNN inference results in Table 3.2 show that for the CNNs with many layers (DenseNet-121 and ResNet-101) comparable performance (throughput)

can be obtained as the mimicked powerful cloud server (NVIDIA GeForce RTX2080).

An observation that can be made in general from our DSE results is that by increasing the number of utilized devices, the per-device memory usage is not always reduced if GPUs are deployed within (some of) the devices. In Table 3.2, this is clearly illustrated by, for example, CNN mappings A and B. These mappings have even higher per-device memory usage when distributing the CNN over, respectively, four and six devices as compared to a 1-Device CPU configuration. The higher memory usage when deploying GPUs is due to the fact that an NVIDIA Jetson Xavier NX device has 8GB memory that is shared between CPU and GPU programs. During the loading phase of CNN models, there will typically be at least two copies of the CNN weights when using the GPU: those from the original model file in the host memory, and those initialized as part of the GPU engine.

3.4.3 *Varying the Number of Edge Devices*

In Figure 3.5, we show the effects on the maximum per-device energy consumption, maximum per-device memory usage, and system throughput when scaling the number of deployed edge devices in the distributed CNN execution. Every bar in Figure 3.5 reflects the best value (energy consumption, memory usage, or throughput) found among all the evaluated mappings, during our DSE experiment, with a specific number of deployed edge devices. This implies that the value reflected by each bar may come from a different Pareto-optimal mapping. For better visualization, all results in Figure 3.5 have been normalized, where the results for a configuration with one edge device are taken as the reference (i.e., these represent the results of the best-found mappings when targeting a single edge device).

From Figure 3.5, we can see that, in general, both the per-device energy consumption and the per-device memory usage can be improved (i.e., reduced) when increasing the number of deployed edge devices. Evidently, this is due to the fact that the workload (the size and/or the number of executed sub-models) on each participating edge device is reduced when increasing the number of edge devices. Moreover, in some cases, the improvement can be significant. For example, for ResNet-101, the maximum per-device energy consumption and maximum memory usage are reduced by around 40% and 80%, respectively, when distributing the CNN over eight edge devices as compared to execution on a single device. Furthermore, the results in Figure 3.5 show that the system (CNN inference) throughput can

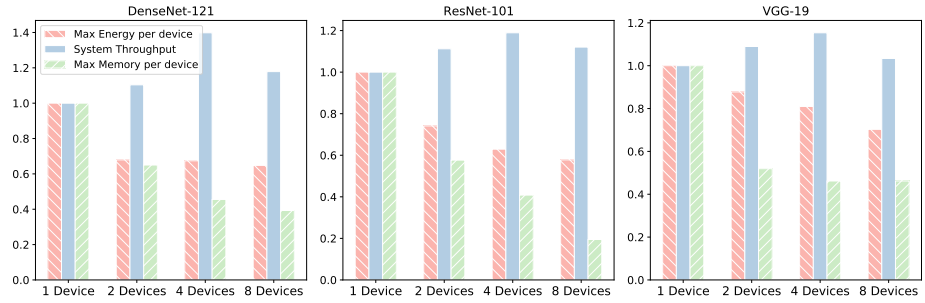


Figure 3.5: System throughput and max energy/memory per device when varying the number of edge devices for three CNNs.

also be improved by means of distributed CNN execution. This is because of the exploitation of pipeline parallelism in the distributed CNN execution. For example, for DenseNet-121, ResNet-101, and VGG-19, the inference throughput increases by up to 38%, 18%, and 18%, respectively when executing the CNN inference on up to four edge devices as compared to a single device. However, the inter-device data communication overheads involved in distributed CNN execution may prevent any further throughput gains, or even cause a slowdown, when scaling the CNN execution to a larger number of edge devices. For example, for all three CNNs, DenseNet-121, ResNet-101, and VGG-19, we see a slowdown in system throughput when scaling the CNN inference from four to eight edge devices.

3.5 DISCUSSION

In our experiments, we have used edge devices that are interconnected using a Gigabit network switch. Evidently, in more realistic edge/IoT settings the connectivity between edge devices might have a lower bandwidth, e.g. using WiFi or other wireless protocols. This would have a detrimental effect on the system throughput objective of distributed CNN inference implementations, possibly leading to more or even purely slowdowns when distributing the inference of a CNN on multiple edge devices. However, we would like to stress that this will not have any impact on the positive effects on (i.e., the reduction of) the per-device energy consumption and per-device memory usage that can always be achieved by distributing CNNs over multiple edge devices.

Additionally, since the Jetson NX boards with 16GB of memory used as edge devices in our experiments are sufficiently equipped for executing

complete CNNs, one could question why distributed execution would be needed. However, in real-world application scenarios, there are often other running application tasks, besides the CNN execution, on an edge device. In such scenarios, the device memory cannot be fully utilized for the CNN execution, and therefore the available memory may be insufficient for CNN-based applications. If CNN models cannot be mapped on a single device because of memory limitations (either due to memory usage of other application tasks on the device or the fact that the device is less capable than the one we used in our experiments and simply has not enough physical memory), then we have to split the CNN model and execute it on multiple collaborative edge/IoT devices.

Another important reason for distributing CNN execution over multiple edge/IoT devices, even if CNN execution on a single edge/IoT device would be feasible, is when the consumed energy by a single (battery-operated) device does not provide enough 'lifetime' for the application mission to be performed. For example, consider an application scenario where a swarm of eight collaborating battery-operated mobile robots has to perform a surveillance mission for 20 hours without recharging the batteries. One of the tasks, among several mission tasks the swarm has to perform, is a continuous on-board CNN-based image processing of a camera-captured video stream using the ResNet-101 CNN model. Every mobile robot in the swarm is equipped with a Jetson NX board (edge device) used for robot control/navigation and for running tasks related to the mission. Let us assume that the Jetson NX board is powered by a battery with a capacity of 18000 mAh and an output voltage of 19 V. On the one hand, if the CNN-based image processing task of the swarm is assigned to and performed by only one of the robots then, with the aforementioned battery capacity, the execution of the ResNet-101 model on the robot's Jetson NX edge device can last only for 15.24 hours, thus the swarm will not be able to accomplish the 20-hour mission without battery recharging. This is because the energy consumption per image of ResNet-101 executed on Jetson NX is 1.031 J, and after processing 1194181 images with a processing time of 45.94 ms per image, the aforementioned battery will be completely discharged. On the other hand, if the CNN-based image processing task of the swarm is assigned to and performed collaboratively by four out of the eight robots in the swarm, i.e., distributing the ResNet-101 CNN model on four Jetson NX edge devices, then the 20-hour mission of the swarm without battery recharging could be accomplished. This is because, according to our results shown in Figure 3.5 for ResNet-101, the distributed ResNet-101 execution

on four edge devices will reduce the energy consumption per device by around 35%, thereby increasing the 'lifetime' of ResNet-101 on a single battery charge with 1.54x to about 23.45 hours.

The real-world application scenarios and example, discussed above, clearly demonstrate the benefits of reducing the per-device memory usage and per-device energy consumption that could be achieved by using our novel framework for distributed CNN inference at the Edge.

3.6 CONCLUSIONS

In this chapter, we have presented AutoDiCE, the first fully automated framework for distributed CNN inference over multiple resource-constrained devices at the Edge. The framework features a unified and flexible user interface, fast CNN model partitioning and code generation, and easy deployment of the CNN partitions on edge devices. By applying the design flow on three representative CNNs, we have evaluated AutoDiCE in terms of efficiency and usefulness in facilitating fast and accurate DSE. The results show that AutoDiCE can easily and rapidly realize a wide variety of distributed CNN implementations on multiple edge devices, achieving improved (i.e., reduced) per-device energy consumption and per-device memory usage. It is worth noting that these improvements are achieved without losing the initial CNN model accuracy because the steps in our framework change neither the CNN layers and their data dependencies nor the values and precision of the CNN parameters.

4

CHAPTER 4

Design Space Exploration (DSE) methods are becoming essential to find a set of optimal CNN mappings subject to one or more design requirements, as the number of different CNN mapping possibilities when deploying a CNN model on multiple edge devices is vast. To facilitate this DSE process, we present an efficient DSE method to find (near-)optimal CNN mappings for distributed inference at the Edge. To deal with the vast design space of different CNN mappings, we accelerate the searching process by proposing and utilizing a multi-stage hierarchical DSE approach together with a tailored Genetic Algorithm as the underlying search engine.

This Chapter is based on the workshop paper:

- **Xiaotian, Guo**, Andy D. Pimentel, and Todor Stefanov. *"Hierarchical design space exploration for distributed CNN inference at the Edge"* [57], in *3rd Workshop on IoT, Edge and Mobile for Embedded Machine Learning (ITEM 2022), part of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, © Springer.

4.1 INTRODUCTION

As discussed in Chapter 1, deploying large CNN models on edge devices collaboratively presents challenges due to the limited resources of these devices, often requiring reliance on cloud services which can introduce issues of data privacy and latency. To address these challenges, collaboratively performing the CNN inference across multiple edge devices has been proposed in Chapter 3. However, such distributed execution of the CNN model inference often needs to take multiple requirements into account, like latency, throughput, resource usage, power/energy consumption, etc. Here, the way how the different CNN layers are distributed and mapped onto the edge devices plays a key role in optimizing/satisfying these requirements. For example, using model-parallelism techniques and mapping CNN layers in a balanced way may reduce the maximum per-device memory footprint or energy consumption. Or, some CNN mappings may generate a balanced data processing pipeline, thereby improving the overall throughput. As CNN models for modern applications are becoming increasingly deep and complex, the number of different CNN mapping possibilities when deploying multiple edge devices, and the various compute resources in each of them, is vast. Efficient Design Space Exploration (DSE) methods are therefore essential to find a set of (near-)optimal CNN mappings subject to one or more design requirements (i.e., objectives).

In this chapter, we present an efficient DSE method to find optimal CNN mappings for distributed inference at the Edge. To this end, we leverage our AutoDiCE framework [32] discussed in Chapter 3 to assess the quality (in terms of inference throughput, memory footprint, and energy consumption) of a particular CNN mapping. To deal with the vast design space of different CNN mappings, we accelerate the searching process by using a multi-stage hierarchical DSE approach together with a tailored Genetic Algorithm (GA) as the underlying search engine. At every stage, we perform DSE at two hierarchical levels. In the first level, we use analytical models inside the GA to approximate each objective function (i.e., throughput, memory, and energy consumption) to avoid relatively long evaluation times through real on-device (i.e., on-board) measurements using our AutoDiCE framework. The near-optimal solutions found in the first level together with Pareto-optimal solutions from a previous DSE stage are utilized as the parents for the second level DSE. In this second level, we evaluate each design point using real measurements taken from AutoDiCE-generated CNN inference implementations to determine the Pareto front for a next DSE stage.

The output of the last DSE stage provides the final Pareto-optimal solutions. Our contributions can be summarized as follows:

- We accelerate the DSE convergence by performing the DSE process in multiple stages where, at each DSE stage, we consider only a specific part of the design space and use as input Pareto-optimal solutions from the previous DSE stage in order to find Pareto-optimal solutions for the next DSE stage;
- We improve the searching efficiency with a tailored chromosome encoding method, thereby scaling down the search space.

4.2 RELATED WORK

To perform CNN inference on a fully distributed system at the Edge, without any cloud involvement, data partitioning [18] or model partitioning [19] is often required. Meanwhile, researchers try to optimize the CNN mapping to improve the inference performance. For example, the methodologies in [58–60] propose efficient algorithms to determine partitioning policies that generate efficient CNN mappings in order to improve the performance of cooperative inference over multiple edge devices. However, these methodologies optimize and evaluate CNN mappings based on analytical models only and consider a limited number of objectives. In contrast, our DSE method optimizes more objectives, and besides analytical models, it uses AutoDiCE to evaluate mappings by real on-device measurements.

Distributed inference of large CNN models typically needs to consider a range of different design requirements, such as latency, throughput, resource usage, power/energy consumption, etc. These requirements/objectives can be conflicting, implying that there usually does not exist a single optimal CNN mapping that satisfies all requirements. Typically multiple solutions, so-called Pareto optimal solutions, co-exist, and the set of all optimal solutions is called the Pareto front. Finding these Pareto-optimal CNN mappings for a given number of edge devices to perform distributed CNN inference under several requirements is the topic of study in this chapter. As discussed in Chapter 2, NSGA-II [5] is a popular approach to perform such a search for Pareto-optimal solutions. For instance, [61, 62] use the NSGA-II to explore the design space to find improved neural network architectures for CNN-based applications. Our DSE method also employs NSGA-II to explore the Pareto-optimal CNN mapping solutions with respect to throughput, maximum memory usage per device, and maximum

energy consumption per device. However, NSGA-II can easily get stuck in so-called dominance resistant solutions [63], that are far away from the true Pareto front. How to search the optimal CNN mappings for distributed inference using NSGA-II, and efficiently find the Pareto front in the huge search space, are the main challenges we try to tackle in this chapter.

4.3 METHOD

Our DSE method utilizes a Genetic Algorithm (GA), namely the NSGA-II algorithm [5] introduced in Section 2.3.1, to search for optimal mappings of (complete) CNN layers to different, distributed edge devices. We assume that each edge device contains a number of internal compute resources (i.e., PEs), like a CPU and GPU, and we map CNN layers directly to these specific PEs within an edge device.

4.3.1 *Fitness Functions*

We use two different fitness functions for the evaluation of the three objectives at every stage in our two-level DSE. The first level DSE applies the analytical models, discussed in Section 2.3.3.1, to approximate the objectives. The second level uses our AutoDiCE framework, introduced in Chapter 3, to evaluate the objectives of distributed CNN inference by real implementations and measurements on the Jetson Xavier NX hardware devices described in Section 2.3.3.2. The first level allows for a rapid DSE search using approximate evaluations based on the analytical models, while the second level facilitates a more detailed and accurate assessment of CNN mappings based on the AutoDiCE framework.

4.3.2 *Multi-stage hierarchical DSE*

Given a trained CNN model with L layers, a layer l_j performs a computation operation in the CNN model such as a convolution (Conv), a matrix multiplication (FC), etc. A mapping $\mathbf{x}$ of the CNN layers onto a total of N PEs is denoted as $\mathbf{x} = [x_1, x_2, \cdots, x_L]$. As explained in Section 2.3, such mapping notation $\mathbf{x}$ is typically encoded with the GA's chromosome where $PE_i, i \in [1..N]$ define the gene types in the chromosome. An example of such encoding, called Naive Encoding (NE), is shown in Figure 4.1. The GA chromosome [PE_1,PE_1,PE_2,PE_2,PE_3,PE_4,PE_4,PE_4] encodes an 8-layer CNN ($L = 8$) mapped onto four PEs ($N = 4$), where layers l_1 and l_2 are mapped

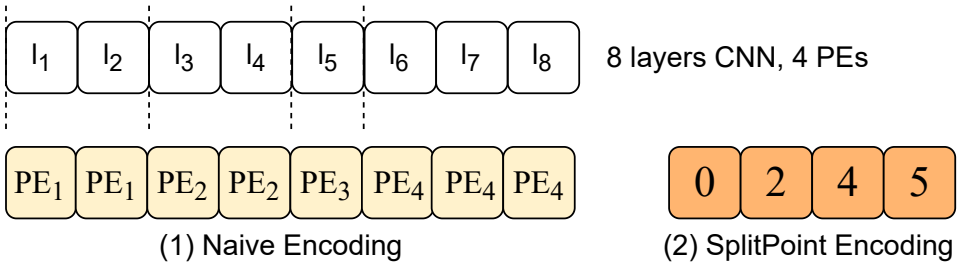


Figure 4.1: Two Chromosome Encoding Methods

on PE_1, l_3 and l_4 on PE_2, l_5 on PE_3, and l_6, l_7, l_8 on PE_4. Such naive encoding for CNN mappings is simple and intuitive but it may require exploration of a huge design space because the space size depends exponentially on the number of layers L in a CNN model and L is typically large. Therefore, in our DSE method, we propose and utilize a tailored chromosome encoding method, called Split Point Encoding (SPE). It encodes points in a CNN model that partition the model into N groups of CNN layers, where each group consists of consecutive layers and is mapped on one PE. In Figure 4.1, the Split Point Encoding example encodes the same mapping as the Naive Encoding example. It can be seen that the 8-layer CNN has four split points, visualized with the vertical dashed lines, at positions 0, 2, 4, and 5 determined by the layer index j. Therefore, the GA chromosome using our SPE method is [0, 2, 4, 5] and it encodes four groups of layers each mapped on one PE as follows: 1) for $j \in (0..2]$, l_j is mapped on PE_1; 2) for $j \in (2..4]$, l_j is mapped on PE_2; 3) for $j \in (4..5]$, l_j is mapped on PE_3; 4) for $j > 5$, l_j is mapped on PE_4. The length of our SPE chromosome is equal to the number of PEs which is N, thus SPE requires exploration of a design space which size depends exponentially on N. Since N is typically much smaller than the number of CNN layers L, our SPE method largely scales down the design space and improves the search efficiency compared to the NE method.

Given a trained CNN model and all edge devices within total N PEs, our DSE method searches for Pareto CNN mappings to optimize the three objectives, i.e. maximum memory usage per device, maximum energy consumption per device, and overall system throughput. In Figure 4.2, we present the general structure of our multi-stage hierarchical DSE method. On the left, the K stages in our DSE workflow are depicted, and on the right, a zoomed-in view of each stage is provided with the two rectangular boxes showing the two hierarchical levels per stage. We accelerate our DSE pro-

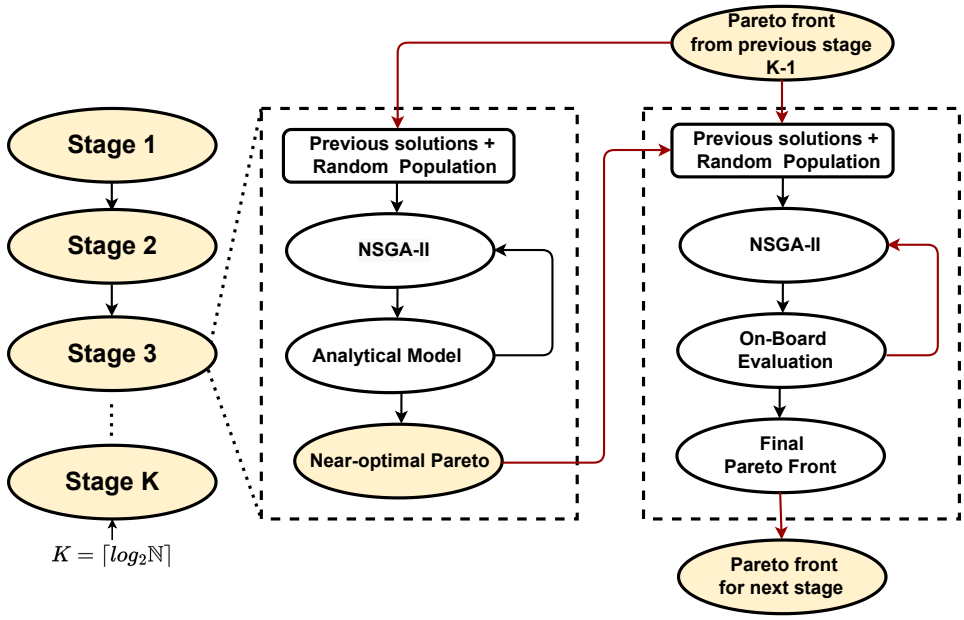


Figure 4.2: The DSE method workflow

cess by splitting it into K different stages, where K is the ceiling value of $\log_2(N)$. At each stage, we perform a two-level DSE. At both levels, the NSGA-II GA is deployed to evolve a population of CNN mappings over multiple generations to search for a Pareto front in terms of the targeted objectives. In the first DSE level, we use the analytical models, introduced in Section 2.3.3.1, inside the GA to approximate each objective function. In the second DSE level, we use real distributed CNN inference implementations generated by AutoDiCE (see Figure 3.1) for evaluation, thereby producing more accurate Pareto solutions as they are based on real (on-board) measurements.

At every DSE stage $k \in [1..K-1]$, we search for optimal CNN mappings on 2^k target PEs. Figure 4.2 shows that to initialize the GA population at stage k, with $k > 1$, the Pareto optimal results found by the previous stage $k-1$ are used. By doing so, we can retain the information of Pareto CNN mappings in previous stages to improve the DSE convergence. Moreover, the second level DSE at each stage also uses the results from the first level of DSE to initialize its population. Finally, the output of the last DSE stage ($k = K$) provides the final Pareto-optimal solutions for N PEs.

4.4 EXPERIMENTAL EVALUATION

In this section, we evaluate the search efficiency of our multi-stage hierarchical DSE method by conducting three DSE experiments and comparing the obtained experimental results in terms of the quality of the found solutions and how this quality changes over time during the DSE process (i.e., the search).

4.4.1 *Experimental setup*

In our three DSE experiments, we search for Pareto-optimal mappings of the popular ResNet-101 [56] CNN model onto a cluster of four edge devices. ResNet-101 has 344 layers with diverse types leading to an immense number of different CNN mappings, i.e., we have to perform the search in a vast design space. Therefore, ResNet-101 is a sufficiently representative model to apply our DSE method on and to demonstrate its merits. We use four NVIDIA Jetson Xavier NX development boards [27] in our cluster (Section 2.3.3.2). As each board has a 6-core CPU (NVIDIA Carmel ARMv8) and a GPU, we have 8 PEs in total in our edge cluster (4 boards with 1 CPU and 1 GPU per board) for the experimental setup. The On-Board Evaluation step in the second level of our DSE method (see Figure 4.2) measures and collects the CNN inference throughput, memory usage per device, and energy consumption per device over 20 CNN inference executions and represents them as average values over these 20 executions.

In the first DSE experiment, referred as 3s-2l-SPE, we utilize our multi-stage hierarchical DSE method as presented in Section 4.3 with 3 stages, 2 levels per stage, and the chromosome is encoded using our SPE method. In the second experiment, referred as 1s-non-SPE, we utilize a classical 1-stage, non-hierarchical DSE method based on the NSGA-II algorithm with our On-Board Evaluation as the fitness function and our SPE as the chromosome encoding method. In the third experiment, referred as 1s-non-NE, we utilize the same DSE method as in the second experiment but we replace SPE with the NE method mentioned in Section 4.3. In all experiments, every CNN layer can be mapped either onto a 6-core CPU or a GPU present in any of the aforementioned four boards. The NSGA-II algorithm is executed with a population size of 100 individuals, a mutation probability of 0.2, and a crossover probability of 0.5. In each DSE experiment, we run the search for optimal mappings for 70 hours and compare the quality of solutions found within these 70 hours.

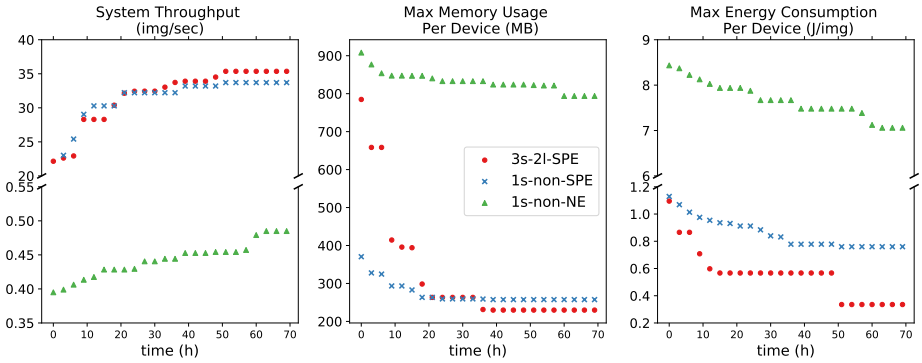


Figure 4.3: Quality of found mappings during the three DSE experiments.

4.4.2 *Experimental results*

Figure 4.3 shows how the quality of the found mappings in terms of the three targeted objectives improves during the search in the three DSE experiments. The results for each objective are plotted in a separate chart where the X-axis represents the search time in hours and the Y-axis represents the objective value in images per second (img/sec) for the CNN inference throughput, in megabytes (MB) for the maximum memory usage per edge device, and in joules per image (J/img) for the maximum energy consumption per edge device. Every point in a chart represents the best-found mapping with respect to the objective at a given point in time.

The results in Figure 4.3 clearly indicate that the 1s-non-NE DSE gets easily stuck in dominance resistant solutions, which means that such DSE cannot find high-quality mappings even after hundreds of generations. In contrast, by replacing the common NE encoding method with our tailored SPE method, the search efficiency is significantly improved as shown in Figure 4.3 where the 1s-non-SPE DSE delivers high-quality mappings for the three objectives after 20 hours. This is because our SPE method ensures that only consecutive CNN layers will be mapped on a PE, thereby scaling down significantly the design space and allowing only exploration of mappings with reduced data communication among PEs. Such mappings are better than less restricted mappings allowed by the NE method.

Finally, comparing the 1s-non-SPE and 3s-2l-SPE results shown in Figure 4.3, we see that by introducing multiple stages and hierarchy in the DSE process, it further accelerates the finding of high-quality mappings.

For example, after 40 hours of search time, our 3s-2l-SPE DSE delivers better mappings for the three objectives than the 1s-non-SPE DSE.

4.5 CONCLUSION

We have presented a novel multi-stage hierarchical DSE method for distributed CNN inference at the Edge. To accelerate the DSE process and improve its efficiency, our DSE method combines analytical models with real on-board measurements to speedup the evaluations of individual design points and utilizes a tailored chromosome encoding method to effectively scale down the explored design space. The method has been experimentally evaluated by searching for optimal distributed mappings of the ResNet-101 CNN model onto an edge cluster of four NVIDIA Jetson Xavier boards. The experimental results show that our multi-stage hierarchical DSE method has significantly improved search efficiency in comparison to a classical one-stage, non-hierarchical DSE method which employs the commonly used, naive chromosome encoding method.

Part II

ROBUSTNESS FOR DISTRIBUTED INFERENCE

The second part of the thesis delves into the robustness of distributed Deep Neural Network (DNN) inference, presenting two pivotal chapters that build upon the AutoDiCE framework discussed in the first part of the thesis. This part addresses a critical challenge inherent to distributed systems, particularly in edge computing environments: the potential unavailability of devices due to failures or unreliable connections.

Chapter 5 introduces the RobustDiCE method, which enhances the robustness of distributed DNN inference by strategically replicating critical neurons across multiple devices. This method ensures continuous and reliable inference even when some devices fail or are temporarily unreachable, focusing on maintaining system performance through balanced neuron dispersion.

Chapter 6 presents the EASTER method, designed to optimize replication strategies for DNNs, particularly large transformer models. EASTER methodically searches for optimal neuron replication ratios within each layer to balance the robustness against device failures with resource utilization and performance, aiming to achieve robust, efficient, and effective distributed inference.

5

CHAPTER 5

Distributing the computations and coefficients of CNN models over multiple edge devices collaboratively has been well studied but these existing works generally do not consider the presence of device failures (e.g., due to temporary connectivity issues, overload, discharged battery, etc. of edge devices). Such unpredictable failures can compromise the reliability of edge devices, inhibiting the proper execution of distributed CNN inference. Therefore, in this Chapter, we present a novel partitioning method, called RobustDiCE, for robust distribution and inference of CNN models over multiple edge devices. Our method can tolerate intermittent and permanent device failures in a distributed system at the Edge, offering a tunable trade-off between robustness (i.e., retaining model accuracy after failures) and resource utilization. We evaluate RobustDiCE using the ImageNet-1K dataset on several representative CNN models under various device failure scenarios and compare it with several state-of-the-art partitioning methods as well as an optimal robustness approach (i.e., full neuron replication). In addition, we demonstrate RobustDiCE's advantages in terms of memory usage, and energy consumption per device, and system throughput for various system set-ups with different device counts.

This Chapter is based on the conference paper and the journal article:

- **Xiaotian, Guo**, Quan Jiang, Andy D. Pimentel, and Todor Stefanov. *"RobustDiCE: Robust and Distributed CNN Inference at the Edge"* [64], in *29th Asia and South Pacific Design Automation Conference (ASP-DAC 2024)*
- **Xiaotian, Guo**, Quan Jiang, Andy D. Pimentel, and Todor Stefanov. *"Model and System Robustness in Distributed CNN Inference at the Edge"* [65], in *Integration, the VLSI Journal*

5.1 INTRODUCTION

As Artificial Intelligence (AI) continues its rapid evolution, convolutional neural networks (CNNs) are becoming increasingly prevalent across a variety of applications [66]. The surge of Internet-of-Things (IoT) devices has also elevated the deployment requirements of CNNs at the Edge. However, the growing complexity and size of CNN models, such as VGG-16 [6], and CoAtNet-6 [45], pose a significant challenge in terms of computing resources for resource-constrained edge devices. As discussed in Chapter 3 and Chapter 4, distributed CNN inference is a desired approach to alleviate the discrepancy between the constrained resources of edge devices and the huge requirements of deploying large CNN models.

However, current partitioning methods assume continuous availability of all involved edge devices that cannot be always guaranteed because an edge device could be temporarily unreachable (especially when edge devices are mobile and use low-power short distance radios for communication) or a device could experience a temporary failure (e.g., due to a discharged battery). Therefore, it is imperative to devise and utilize partitioning methods for distributed CNN inference with robustness in mind.

In this chapter, we present a novel partitioning method, called **RobustDiCE**, for robust distribution and inference of CNN models over multiple edge devices. RobustDiCE features both *system robustness*, i.e., CNN inference can continue execution even if one or more edge devices fail to function properly, and *model robustness*, i.e. preserving the inference accuracy of the CNN model as much as possible when some of the intermediate CNN inference results are lost due to failed devices. We improve the system robustness by implementing a decentralized architecture that incorporates a robust fault handler for reliable execution. The fault handler's internal timeout mechanism, regulated by periodic heartbeats [67], prevents system deadlocks and improves resilience against potential device failures or network disruptions. Moreover, we address the model robustness challenge by evaluating the relative importance of each neuron in the CNN model and then partitioning these different neurons of each CNN layer into different groups (to be mapped to the various edge devices) as 'evenly' as possible. Our main novel contributions can be summarized as follows:

- Based on the importance criterion of different neurons in each CNN layer, a new partitioning method is proposed to preserve the model accuracy as much as possible against device failures. This new method combines *partial* neuron replication and *importance-aware* neuron clus-

tering to achieve CNN model robustness. It also provides a tunable trade-off between robustness (i.e., retaining model accuracy after failures) and resource utilization.

- We evaluate our novel partitioning method using the ImageNet-1K dataset on several representative CNN models under pessimistic device failure scenarios. We compare it with a number of state-of-the-art (partitioning) approaches, including the CDC method [68] leveraging neuron replication to increase robustness and an ideal robustness approach utilizing full neuron replication.
- We demonstrate our method's superiority in terms of memory usage and energy consumption per device, and system throughput under different system configurations.

5.2 RELATED WORK

Processing the input data collaboratively by utilizing multiple edge devices helps to mitigate the resource limitations of a single edge device in terms of available memory, energy budget, etc. Model partitioning [21] and data partitioning [18] mentioned in Chapter 3 are two common methods for implementing distributed CNN inference. Studies [18–23, 69] try to optimize the CNN partitioning to improve inference performance under different conditions such as network bandwidth, neural network topology, and hardware specifications [23].

However, all these methods assume that the involved computing edge devices and communication links between them are always available and work properly in a long-running CNN inference task (e.g., in a continuous stream of input data). They are not designed to be robust against temporary or permanent unavailability/failures of devices/links.

System robustness in distributed CNN inference refers to the ability of a system to continue functioning correctly, or to provide graceful degradation, even in the presence of hardware or software failures (such as the unavailability of system resources, communication failures, invalid or excessive input data, etc.) [70]. The majority of studies on robust computing focus on replication [71–73], redundancy [74, 75], error correction [68, 76], checkpoint recovery [77], and fault tolerance techniques [78]. For example, [71–73] keep the results of multiple computing nodes up to date and consistent through replication. [74, 75] replicate computing nodes of the distributed system to provide multiple identical instances as backups

in case of a node failure. However, full replication of input data and CNN layers (with their vast numbers of weights/biases) for CNN inference is not feasible for resource-constrained edge devices. Moreover, the usage of many redundant hardware devices is expensive and not suitable in all cases. [76] uses error correcting codes (ECCs) to protect the weights from perturbations while [68] applies a coded distributed computing (CDC) method for fast recovery of output results from a certain number of node failures. However, error correction methods introduce extra computations on the edge devices which may be difficult to facilitate given the limited available resources. [77] provides a checkpoint recovery mechanism for "continued execution" where the deep learning implementation continues to execute by utilizing the remaining set of computing nodes. However, the recovery method increases the physical memory usage, and the recovery time is limited to the filesystem I/O bottleneck of the edge devices. Unlike the previous works, our work focuses on robust, distributed CNN inference with the goal of reducing the computational and memory resource usage per edge device to better match the limited resources of edge devices.

Model robustness of distributed CNN inference concerns the property of a model of being resilient in terms of inference accuracy to the failure of physical computing nodes due to power outages, unstable inter-node connections, other hardware/software failures, etc. In distributed CNN inference, the missing neurons on those failed nodes may result in a significant accuracy drop [68] of a CNN model. To alleviate the influence of node failures on the CNN inference accuracy, several failure-aware retraining methods [79–84] for CNNs have been developed. For example, if layer connections of CNN models are split, the forward process of the CNN inference cannot continue because of the presence of failed nodes. DeepFogGuard [85] establishes skip hyperconnections to skip certain failed physical nodes during the retraining process. ResiliNet [82] introduces failout to simulate physical node failure conditions during retraining. [81] retrains CNN models to be resilient to packet loss in a lossy IoT network. The retraining process adds dropout on certain CNN layers but this cannot guarantee that the model accuracy is preserved. When the dropout rate is too high, thereby simulating a high percentage of node failures, the retraining model may result in under-learning which causes a significant decrease in its accuracy. In addition, all these retrained models are designed to be aware of only specific failures such as communication failures between two CNN layers, certain node failures in a pipeline multi-node inference, etc. Moreover, CNN retraining requires a large amount of data that may not be always accessible

for an end user of a pre-trained CNN model to perform retraining before the deployment in an unreliable environment. As most state-of-the-art pre-trained models directly available to an end user for deployment are not failure-aware, our RobustDiCE method can be easily applied to partition these pre-trained models to achieve system and model robustness without any retraining, without assuming specific types of failures, and without suffering from accuracy degradation due to parameter changes (e.g., by adding dropout on CNN layers) in the neural networks. Moreover, our robustness method can be seen as complementary to these retraining approaches, i.e., if we would apply our method to the aforementioned retrained models, we can further improve their robustness against node failures.

To summarize, performing robust inference on distributed edge devices is vital. Existing robustness methods suffer from extra computing resource requirements, time-consuming retraining, accuracy degradation, etc. In contrast, our method RobustDiCE is designed to guarantee robustness under limited resources of edge devices. In addition, our method is a post-training technique that provides robustness without the need for CNN model retraining. Furthermore, we have implemented and tested our robust distributed CNN inference on real physical edge devices. Both system robustness and model robustness are provided by our method and verified via experimental results.

5.3 BACKGROUND AND MOTIVATION

In this section, we provide some specific background information and a motivational example to understand our novel CNN partitioning method for robustness.

Generally, state-of-the-art partitioning methods, such as discussed in [21], do not consider robustness as they do not consider the fact that different neurons/filters in CNN layers have different *importance*, thereby causing various effects on the inference accuracy of a CNN model, particularly those neurons with larger values [86]. The relative *importance* of a neuron in a CNN layer can be measured by calculating metrics such as the l_1-norm [87], l_2-norm [88], etc. To partition a CNN layer with robustness in mind, it is essential to find an effective way to group and distribute its neurons/filters over computing nodes as evenly as possible in terms of *importance*.

To clarify this statement, we use the simple example, shown in Figure 5.1, where we consider a convolution layer with five filters/neurons denoted as n_1 to n_5. We want to partition these neurons over three computing nodes.

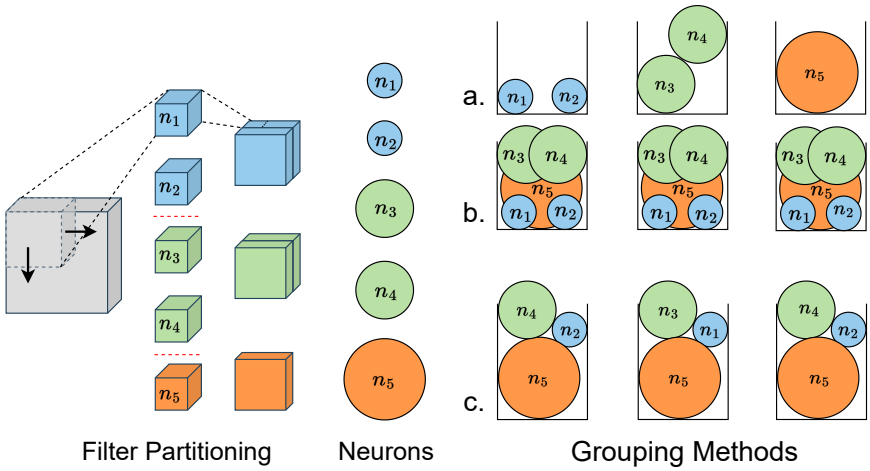


Figure 5.1: Typical vs. Robust Partitioning

In this example, the importance score s_j of each neuron n_j is measured by calculating the l_1-norm, i.e., taking the filter corresponding to neuron n_j with shape $C_{in} \times k \times k$ (where k denotes the kernel size of the filter and C_{in} the number of input channels), we calculate the sum of absolute values of all the weights in the filter and its bias. In the middle and the right part of Figure 5.1, we visualize the importance s_j of each neuron n_j by the size of the circle representing the neuron, i.e., neuron n_5 has the highest importance whereas n_1 and n_2 have the lowest importance.

As shown in Figure 5.1(a), a partitioning method without robustness in mind (i.e., no consideration of the neurons' importance s_j) splits the five neurons into three groups (visualized by the three colors in Figure 5.1) and the groups are distributed over the three nodes. Such distribution reduces computational resources per node because the layer workload is split over the nodes. However, this distribution is not robust at all because if, for example, the third node fails, which runs the most important neuron n_5, then the inference accuracy will decrease significantly.

To maximize the robustness, well-known modular redundancy methods can be applied as shown in Figure 5.1(b). Here, we replicate all neurons over the three nodes, thereby achieving maximum robustness against failures because even if one or two nodes fail then the remaining available node will run all the neurons without a decrease in inference accuracy. However, this significantly increases the resource requirements (e.g., memory and energy consumption) for each node. Moreover, this full replication approach might be infeasible for resource-constrained nodes due to the limitations with

respect to their computational or memory resources and the possible energy budget of an edge device.

The two example scenarios, illustrated in Figure 5.1(a) and (b), clearly show that using existing, robustness-unaware partitioning methods or modular redundancy methods in isolation cannot provide efficient, robust distributed CNN inference on multiple resource-constrained edge devices. For this reason, in this chapter, we propose a novel method, explained in detail in Section 5.4, which *combines replication and importance-aware partitioning* to achieve high and tunable robustness in an efficient way for distributed CNN inference. The result of applying our method to our simple example is illustrated in Figure 5.1(c). The basic idea is that some (not all) neurons in a CNN layer are replicated and all neurons (initial and replicas) are partitioned into groups and distributed evenly over the nodes based on their *importance*.

The advantage of this partitioning method is that if either the first or third node fails, the remaining nodes can still run all the neurons, thereby preserving inference accuracy. If the second node fails, the critical neuron n_5 still remains, limiting the accuracy degradation. Therefore, we can achieve comparable robustness to the scenario in Figure 5.1(b), but with reduced computational resource requirements, as not all neurons are replicated or run on each node.

5.4 THE ROBUSTDICE METHOD

Our method RobustDiCE features both *system robustness*, i.e., CNN inference can continue to execute or recover even if one or more computing nodes and/or communication links between them fail to function properly and *model robustness*, i.e., when some of the CNN neurons and/or their input/output data are lost due to the failed nodes/links, the inference accuracy of the CNN model is preserved as much as possible. In Section 5.4.1, we briefly outline our decentralized computing framework for CNN inference that supports system robustness. We support the model robustness by applying the new partitioning method, presented in Section 5.4.2, that combines partial neuron replication and importance-aware neuron grouping and distribution over multiple nodes.

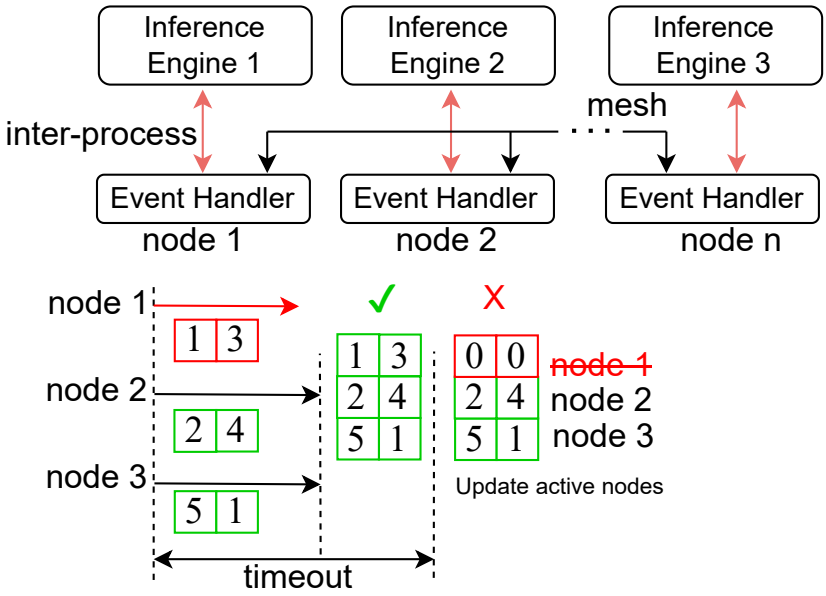


Figure 5.2: Our Decentralized Computing Framework

5.4.1 *Decentralized Computing Framework*

We have devised and implemented a decentralized framework for distributed computing and communication. Such an infrastructure is crucial to ensure that the distributed CNN inference can be executed collaboratively and properly, even in the case of node/link failures. A very high-level overview of our decentralized computing framework for distributed CNN inference is shown in Figure 5.2. In this framework, all nodes are interconnected in a peer-to-peer fashion through an N-to-N mesh topology. On each computing node, there are two running processes: *inference engine* and *event handler*. The inference engine mainly takes care of the CNN computations, whereas the event handler is responsible for handling events such as interconnections, heartbeat and data synchronization between nodes, node failures, node management, etc. The two processes communicate with each other directly through the inter-process communication mechanism.

In the N-to-N mesh topology, each node acts as both a client and a server, thereby allowing for direct communication between nodes without centralized servers or coordinators. The client on each node connects with a set of servers on other nodes, and the server on each node is capable of replacing other servers if necessary. This design ensures that a failure of a node or communication link will not disrupt the entire system. The client in the

event handler of each node continuously sends regular messages, known as heartbeats, to servers on other nodes to confirm their operational status. As shown in the Figure 5.2, we typically set a relatively large value for the timeout to ensure that the data synchronization between the event handlers of these nodes is completed before reaching the timeout. Once data synchronization is completed, the event handler will activate the inference engine to proceed with the corresponding computations. However, if a node fails to send out a heartbeat within a predefined timeout, other nodes assume the unresponsive node has failed. Then the server would remove the inactive or failed nodes and update its list of active nodes.

For example, when all nodes have completed their assigned computations, the partial results, illustrated as values in green boxes on each node, need to be synchronized. If communication proceeds without any failures, all nodes will receive the correct full results, as shown in the 2x3 green box configuration. However, if node 1 encounters a failure and cannot send its partial results (values 1 and 3) to the other nodes, these nodes will end up waiting for Node 1. If Node 1 is unable to send a heartbeat within the predefined timeout period due to its failure, the event handlers on the other nodes will detect this failure and disconnect from Node 1.

Subsequently, all nodes, except Node 1, will proceed to zero out the vector content related to node 1's contributions, indicated by the red box. Although the data synchronization results on these nodes will be incorrect without node 1's input, they will still proceed to activate the inference engine to continue processing tasks. This setup allows each node to manage failures independently, ensuring that both computation and communication persist without significant disruption, even in the event of a node failure.

By enabling each node to perform its computation and communication independently, our decentralized framework offers robust and scalable computation and communication for distributed CNN inference. This approach ensures that the system remains functional and efficient even in the face of individual node or link failures.

5.4.2 *Robust Partitioning*

In this section, we present our new partitioning method which achieves CNN model robustness by combining importance-aware neuron clustering and grouping with partial neuron replication in order to evenly distribute the neurons in a CNN model over multiple nodes. The partitioning method

is applied layer-wise on every layer until the whole CNN model is partitioned. The general layer-wise partitioning procedure is outlined in Algorithm 1. It accepts as inputs a set of computational layers L from the CNN model and their coefficients W as well as the total number of computing nodes ND across which the CNN model will be distributed. Additionally, a set T of threshold values corresponding to layers in L is provided as another input. The threshold values serve as specific criteria for identifying similar neurons in terms of importance and subsequently making neuron grouping decisions based on the similarity. The output of Algorithm 1 is set P of neuron partitions. Every partition $P_i = \{p^1, ..., p^{ND}\} \in P$ determines how the neurons in layer $l_i \in L$ are distributed across the specified number of computing nodes ND.

The goal of Algorithm 1 is to evenly distribute the neurons n_j of every layer $l_i \in L$ over ND nodes (i.e., devices) in terms of importance. For example, applying Algorithm 1 (Lines 3–31) to the convolution layer with the five neurons n_1 to n_5 shown in Figure 5.1 and setting $ND = 3$, the output P of the algorithm is the partition illustrated in Figure 5.1(c). Algorithm 1 consists of three main steps performed on each layer $l_i \in L$.

In Step 1 (Lines 4–10), we first include each neuron $n_j \in l_i$ into a separate group G_j which is stored in the set of groups G (Line 6). Then, we calculate three importance scores for n_j from three different perspectives. The first score s_j^1 (Line 8) is the l_1-norm [87] which is a magnitude-based approach, widely used in CNN pruning techniques, to compute neuron importance based on the sum of its absolute weights and bias. The second importance score s_j^2 (Line 9) of n_j is computed by summing the sensitivity scores of all its connections with other neurons. We use the Taylor expansion approach [89] to obtain the connection sensitivity scores through the gradient in the propagation process [90]. The third score s_j^3 (Line 10) assesses the neuron importance by employing the Jensen-Shannon divergence [91] denoted as JSD. A larger change in the CNN output probability distributions $\mathbf{y}$, induced by removing neuron $n_j \in l_i$, indicates that n_j is more important. Instead of using a single importance score only, set $S_j = \{s_j^1, s_j^2, s_j^3\}$ of the three different scores enables a more comprehensive evaluation of the neuron importance because it performs a three-dimensional assessment of the importance, thereby facilitating a more effective clustering of neurons (as will be demonstrated in our experiments).

In Step 2 (Lines 11–21), Algorithm 1 takes the initial set of groups G created in Line 6, where each group contains only one neuron $n_j \in l_i$, and clusters these 1-neuron groups into a *new* set of groups G where any

Algorithm 1: Robust Partitioning

Input : Set of layers L; Number of nodes ND;
Set of layer coefficients $W = \{W_1, ..., W_{|L|}\}$;
Set of threshold values $T = \{t_1, ..., t_{|L|}\}$;
Output: Set of neuron partitions $P = \{P_1, ..., P_{|L|}\}$;

$P \leftarrow \emptyset$
for $l_i \in L$ **do**
  // Step 1: neuron importance scores
  $G \leftarrow \emptyset$
  for $n_j \in l_i$ **do**
    Create G_j; $G_j \leftarrow G_j + n_j$; $G \leftarrow G + G_j$
    Create $S_j = \{s_j^1, s_j^2, s_j^3\}$
    $s_j^1 = \sum_{h=1}^{k_h} \sum_{w=1}^{k_w} \sum_{c=1}^{C_{in}} |W_j^{c,h,w}| + |b_j|$
    $s_j^2 = \sum_{h=1}^{k_h} \sum_{w=1}^{k_w} \sum_{c=1}^{C_{in}} |\frac{\partial y}{\partial W_j^{c,h,w}} \cdot W_j^{c,h,w}| + |\frac{\partial y}{\partial b_j} \cdot b_j|$
    $s_j^3 = \text{JSD}(\mathbf{y}_{\text{complete}} \,\|\, \mathbf{y}_{\text{removing neuron } n_j})$
  // Step 2: neuron clustering
  for $G_z \in G$ **do**
    for $G_q \in G - G_z$ **do**
      $d_{max} = 0$
      for $n_j \in G_z$ **do**
        for $n_o \in G_q$ **do**
          $d(n_j, n_o) = \sqrt{\sum_{a=1}^{3} (s_j^a - s_o^a)^2}$
          if $d(n_j, n_o) > d_{max}$ **then**
            $d_{max} = d(n_j, n_o)$
      if $d_{max} < t_i$ **then**
        $G_z \leftarrow G_z + G_q$
        $G \leftarrow G - G_q$
  // Step 3: round-robin distribution
  Create $P_i = \{p^1, ..., p^{ND}\}$; $p^1 \leftarrow \emptyset, ..., p^{ND} \leftarrow \emptyset$
  for $G_o \in G$ **do**
    if $(|G_o| \mod ND) \neq 0$ **then**
      for $j \in [1, ND - (|G_o| \mod ND)]$ **do**
        Create $n_{|G_o|+j}$ = REPLICA($n_j \in G_o$)
        $G_o \leftarrow G_o + n_{|G_o|+j}$
    for $n_j \in G_o$ **do**
      $r = (j \mod ND) + 1$; $p^r \leftarrow p^r + n_j$
  $P \leftarrow P + P_i$
return P

group may contain multiple neurons with similar importance. To this end, the following two actions are performed iteratively for every two groups $G_z \in G$ and $G_q \in G - G_z$. First, the largest distance d_{max} between the

neurons in G_z and G_q is determined in Lines 13–17. Initially, d_{max} is set to zero. Then, for every pair of neurons $n_j \in G_z$ and $n_o \in G_q$, the Euclidean distance $d(n_j, n_o)$ between n_j and n_o in the three-dimensional importance score space (s^1, s^2, s^3) is computed in Line 16. If $d(n_j, n_o)$ is greater than d_{max}, then d_{max} is updated with $d(n_j, n_o)$ in Line 18.

Second, if d_{max} is below a given threshold value $t_i \in T$ then the neurons in G_z and G_q are merged (Line 20) into one group G_z because they are considered similar in terms of importance, and group G_q is removed from set G in Line 21. The threshold value t_i affects the result of the neurons clustering in Step 2. For example, a small t_i would result in set G having many groups with a few neurons per group. If t_i is too small then every group in G will contain only one neuron, thereby "forcing" the following Step 3 in Algorithm 1 to perform full replication of all neurons, thus maximizing the robustness at the expense of high resource requirements per node in the distributed system. In contrast, a large t_i would result in a few groups with many neurons per group. If t_i is too large then all neurons would be clustered into one group, thereby "forcing" Step 3 to perform very limited or no replication of neurons which could lead to a significant reduction of the robustness. Recall that a set T of threshold values t_i is given as an input to Algorithm 1, thus an optimal set of such values could be determined by integrating Algorithm 1 in a design space exploration (DSE) procedure with multiple optimization objectives including distributed CNN inference accuracy, energy and resource requirements per node in the distributed system, and system performance.

Finally, in Step 3 (Lines 22–30), Algorithm 1 distributes all neurons n_j in every group $G_o \in G$ across a number of nodes ND in a round-robin fashion (Lines 28–29). If the number of neurons in group G_o is not a multiple of the number of nodes ND then some neurons in the group are replicated (Lines 24–27) in order to increase the neuron number to the closest multiple of the number of nodes before the round-robin distribution. Such round-robin distribution can guarantee that every node runs the same number of similarly important neurons from a group, thereby providing CNN model robustness by reducing the CNN inference accuracy degradation in the event of failures in the distributed system.

5.5 EVALUATION OF THE ROBUSTDICE METHOD

In this section, we present a range of experiments demonstrating the merits of RobustDiCE in terms of achieved robustness and resource utilization per node/device in a distributed system performing CNN inference.

5.5.1 *Experimental Setup*

We implement RobustDiCE and apply it to the following distributed system configurations and real-world CNNs, and considering the following device failure scenarios.

CNNs and System Configurations: We experimented with three CNNs, namely AlexNet [92], VGG16-BN [6], and ConvNext-Tiny [93], taken from the TorchVision library. Given their widespread use in image classification and their diversity in layer types, operation counts, and memory requirements for weights, we consider these CNNs to be representative targets to demonstrate the merits of our method. By applying RobustDiCE, every CNN is distributed for inference on three system configurations: one with four edge devices (**SysConf4D**), one with three devices (**SysConf3D**), and one with two devices (**SysConf2D**). Our evaluation cluster contains eight NVIDIA Jetson Xavier NX boards described in Section 2.3.3.2.

Device Failure Scenarios: For each of the aforementioned CNNs, we consider three scenarios.
Scenario A: The CNN is distributed for inference on system configuration **SysConf4D** where 1 device fails (**1D-Fail**), 2 devices fail (**2D-Fail**), or 3 devices fail (**3D-Fail**).
Scenario B: CNN on **SysConf3D** where **1D-Fail** or **2D-Fail**.
Scenario C: CNN on **SysConf2D** where **1D-Fail**.
Under every scenario with a different number of failing devices, we evaluate the preserved Top-1 accuracy on the ImageNet-1K test dataset when the CNN is distributed using our RobustDiCE method. We compare RobustDiCE to state-of-the-art robustness-unaware partitioning that performs filter and layer output partitioning, referred to as *LOP* [21], as well as the robustness-aware CDC method from [68]. In addition, we also show the Top-1 CNN accuracy results under an ideal scenario, called `Optimal`. This `Optimal` scenario assumes that in system configurations **SysConf4D**, **SysConf3D**, and **SysConf2D** no devices fail or all CNN neurons are replicated on every device in order to have quadruple (QMR), triple (TMR), and dual (DMR) modular redundancy, thus achieving maximum robustness.

Table 5.1: Top-1 accuracy (1D-Fail case in SysConf4D)

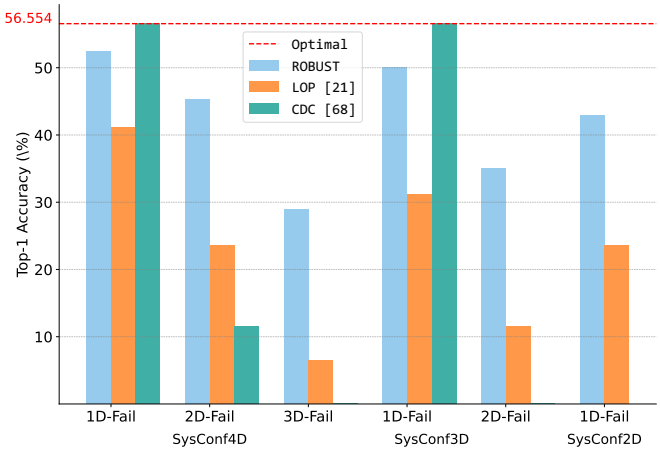
Importance Scores	AlexNet (%)	VGG16_BN (%)	ConvNext_Tiny (%)
s^1	43.718	60.426	76.618
s^2	43.642	58.920	75.904
s^3	43.432	59.942	76.134
$s^1 + s^2$	51.268	69.152	76.678
$s^1 + s^3$	51.658	71.736	76.580
$s^2 + s^3$	51.250	67.360	76.572
$s^1 + s^2 + s^3$	**52.396**	**72.500**	**76.820**

By continuously providing 1000 images as an input data stream for the distributed CNN inference, we measure the system performance in images (frames) per second (FPS), memory usage per device in megabytes (MB), and energy consumption per device in joules per image (J/img) of the distributed CNN inference for the different system configurations. We measure the overall latency in processing the 1000 images and compute averaged FPS as throughput. The energy consumption per device, including CPUs, GPU, communication cost, etc., is obtained through a sampling thread reading power values from the INA3221 monitor on the NVIDIA Jetson Xavier NX board. The memory usage per device is reported directly by the executed CNN code itself during the CNN inference.

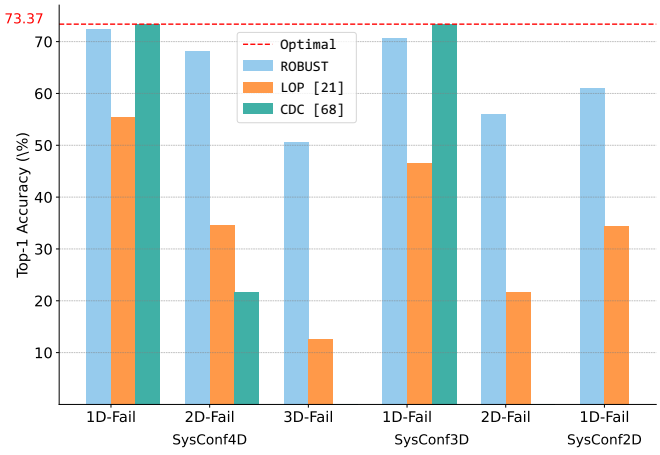
5.5.2 *Experimental Results*

Ablation Study of Importance Scores: To substantiate the efficacy of using multi-dimensional importance evaluation for neuron clustering in Algorithm 1, we carried out an ablation study with various combinations of importance scores (s^1, s^2, s^3). We list the Top-1 accuracy of the 1D-Fail case for the SysConf4D system configuration in Table 5.1 and the other failure scenarios show similar results. It is clear that the combination of all three scores preserves the Top-1 accuracy (model robustness) the best under the 1D-fail scenario for all three models: 52.396% (AlexNet), 72.500% (VGG16_BN), and 76.820% (ConvNext_Tiny). These findings confirm the potential for enhancing model robustness in distributed CNN inference using the combination of multiple importance scores.

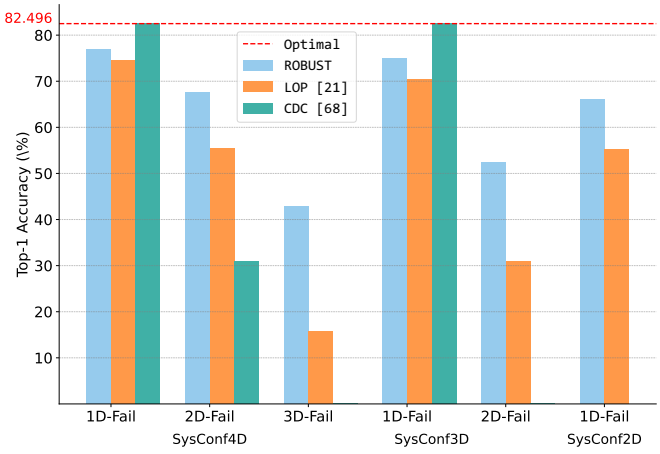
Model Robustness Comparison: The results, obtained with the experimental setup described in Section 5.5.1, are presented in Figure 5.3 and



(a) AlexNet



(b) VGG16-BN



(c) ConvNext-Tiny

Figure 5.3: CNN Model Robustness under different Device Failure Scenarios

Table 5.2. For every CNN model, we show a graph where the X-axis represents the considered scenarios with a different number of failing devices, and the Y-axis indicates the evaluated Top-1 CNN model accuracy. For every scenario and number of failing devices, we plot a bar for the RobustDiCE results (blue bar), LOP results (orange bar), and CDC results (green bar). In addition, the horizontal dashed (red) line shows the accuracy under the `Optimal` scenario.

Looking at the blue and orange bars in Figure 5.3, we observe that RobustDiCE consistently delivers higher Top-1 accuracy compared to the state-of-the-art but robustness-unaware LOP partitioning method. This clearly demonstrates the superiority of our method in terms of CNN model robustness. Taking Figure 5.3(a) as an example, the Top-1 accuracy of AlexNet under the `Optimal` scenario is 56.55% which is our reference point. When a system configuration experiences device failures as in Scenario A, our RobustDiCE method delivers a Top-1 accuracy of 52.40%, 45.28%, and 28.93% for cases 1D-Fail, 2D-Fail, and 3D-Fail, respectively. In contrast, the LOP method exhibits more significant drop in accuracy, namely 41.07%, 23.50%, and 6.42% for the same device failure cases. A similar trend can be observed for VGG-16BN and ConvNext-Tiny in Figure 5.3(b) and (c), respectively. Here, it is important to note that we have used an optimistic device failure scenario for LOP, i.e., devices with the least important groups of neurons fail.

Comparing our RobustDiCE method (orange bars) with the CDC method (green bars), we see that CDC is capable of perfectly handling a single device failure due to its approach of using actor replication and a spare node. However, the CDC method cannot handle multiple device failures, resulting in very low accuracy (much lower than RobustDiCE) or even complete failure (0% accuracy) when all but one devices fail.

Looking at Figure 5.3 and comparing the Top-1 accuracy delivered by RobustDiCE with the reference accuracy under the `Optimal` scenario, we observe that our method does not maintain the reference accuracy level in the event of device failures. The reason is that, in this experiment, we set threshold values $t_i \in T$ discussed in Section 5.4 to be greater than 0 with our prior experience. Because of this, our method does not replicate all CNN neurons on every device, thereby trading off CNN model robustness (loss of Top-1 accuracy) for reduced system resource utilization. This tradeoff could be tuned by changing the t_i values. Moreover, if all t_i values are set to 0 then our method will maintain Top-1 accuracy at the same level as under the `Optimal` scenario. Under this scenario, all CNN neurons

Table 5.2: System performance and resource utilization

Network	System Configuration	Max. per-device Energy (J/img)	System Throughput (FPS)	Max. per-device Memory (MB)
AlexNet	QMR/TMR/DMR	0.179	46.255	150.914
	CDC-SysConf3D	0.165	43.670	94.117
	CDC-SysConf4D	0.157	45.587	78.852
	Robust-SysConf2D	0.159	48.214	99.254
	Robust-SysConf3D	0.148	50.045	80.777
	Robust-SysConf4D	0.142	51.219	72.801
VGG16-BN	QMR/TMR/DMR	0.850	10.744	429.215
	CDC-SysConf3D	0.809	10.634	313.688
	CDC-SysConf4D	0.799	10.485	272.293
	Robust-SysConf2D	0.826	10.761	328.426
	Robust-SysConf3D	0.799	10.993	295.086
	Robust-SysConf4D	0.779	11.078	267.395
ConvNext-Tiny	QMR/TMR/DMR	0.308	28.223	88.895
	CDC-SysConf3D	0.307	27.107	69.129
	CDC-SysConf4D	0.297	28.248	59.961
	Robust-SysConf2D	0.301	28.044	76.465
	Robust-SysConf3D	0.296	28.415	65.203
	Robust-SysConf4D	0.288	29.034	58.090

are replicated on every device in order to have quadruple (QMR), triple (TMR), or dual (DMR) modular redundancy, thus achieving maximum robustness. However, achieving this maximum robustness is at the expense of higher memory usage and energy consumption per device compared to the resource utilization, imposed by our method, when trading off robustness against utilization. This statement is supported by the resource utilization results in Table 5.2. In this table, for every CNN, we show the maximum per-device memory usage (Column 5), the maximum per device energy consumption (Column 3), and the overall system throughput (Column 4) for the three system configurations SysConf4D, SysConf3D, and SysConf2D with our RobustDiCE method and the CDC method as well as for the QMR / TMR / DMR configuration associated with the `Optimal` scenario.

System Performance: Considering the memory usage numbers for AlexNet, shown in Column 5, we see that the replication of all neurons on every device in system configuration QMR / TMR / DMR requires about 150 MB of memory per device. In contrast, our RobustDiCE method significantly reduces the required memory per device, i.e., with 51.76% for system configuration SysConf4D, with 46.47% for SysConf3D, and with 34.23% for SysConf2D. Significant memory reduction trends can be observed in Column 5 for VGG16-BN and ConvNext-Tiny as well. The memory usage

numbers for CDC show that this method reduces the memory footprint in comparison to the all-neuron replication method (QMR / TMR / DMR) but still has higher memory usage compared to RobustDiCE.

The energy consumption per device is also reduced by RobustDiCE as compared to applying all-neuron replication to achieve CNN model robustness. For example, Column 3 in Table 5.2 shows that our method applied on SysConf4D achieves an effective energy reduction over the all-neuron replication method (QMR / TMR / DMR), i.e., 20.67% reduction for AlexNet, 8.35% for VGG16-BN, and 6.49% for ConvNext-Tiny. The CDC energy results again show an improved behavior compared to QMR / TMR / DMR but are inferior to the results from RobustDiCE.

Finally, as shown in Column 4 of Table 5.2, RobustDiCE slightly improves the system throughput for almost all CNNs and system configurations as compared to QMR / TMR / DMR (except for SysConf2D on ConvNext-Tiny). For CDC, on the other hand, the system throughput is generally lower than QMR / TMR / DMR and RobustDiCE.

We note, however, that the system throughput of distributed CNN inference is highly dependent on the quality of the network interconnecting the devices in the system. In our experiments, we have used a Gigabit network switch. Evidently, in other Edge/IoT settings, the connectivity between devices may have a lower bandwidth, e.g., using WiFi or other wireless protocols. Thus, our RobustDiCE method cannot always guarantee system throughput improvements but it can guarantee memory usage and energy consumption reductions per device.

5.6 CONCLUSIONS

This chapter presented RobustDiCE, a robust partitioning method for distributed CNN inference at the Edge that preserves the model accuracy as much as possible against device/link failures. Several CNN experiments demonstrated that RobustDiCE can retain the CNN model accuracy after failures much better as compared to the state-of-the-art partitioning methods. We have also shown the advantages of our RobustDiCE method over the optimal robustness approach and CDC method in terms of memory usage per device, energy consumption per device, and system throughput.

6

CHAPTER 6

*Prevalent large deep learning models, particularly large transformer models, present significant computational challenges for resource-constrained devices at the Edge. While distributing the workload of deep learning models across multiple edge devices has been extensively studied, these works typically overlook the impact of failures of edge devices. Unpredictable failures, due to, e.g., connectivity issues or discharged batteries, can compromise the reliability of inference serving at the Edge. In this chapter, we introduce a novel method, called **EASTER**, designed to learn robust distribution strategies for transformer models against device failures that consider the trade-off between robustness (i.e., maintaining model functionality against failures) and resource utilization (considering memory usage and computations). We evaluate EASTER with three representative transformers – ViT, GPT-2, and Vicuna – under device failures. Our results demonstrate EASTER's efficiency in memory usage, and possible end-to-end latency improvement for inference across multiple edge devices while preserving model accuracy as much as possible under device failures.*

This Chapter is based on:

- **Xiaotian, Guo**, Quan Jiang, Yixian Shen, Andy D. Pimentel, and Todor Stefanov. *"EASTER: learning to split transformers robustly at the Edge"* [94], in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2024*

6.1 INTRODUCTION

As Artificial Intelligence (AI) continues to evolve rapidly, transformer models are increasingly prevalent in various applications [95]. Advanced pre-trained models such as BERT and GPT-4 [96] have spurred a range of novel

tools, including Copilot and ChatGPT. Typically, these models are executed on high-performance clusters with hundreds of GPUs, available as cloud services. However, the rise of Internet-of-Things (IoT) devices has driven a demand for deploying transformer-based tools at the Edge. Deploying these tools on edge or IoT devices offers significant advantages in terms of privacy, local processing requirements, and energy efficiency. For example, a network of IoT devices in smart healthcare systems [97] within a hospital or a home setting such as wearable health monitors, bedside monitors, and portable diagnostic devices are equipped with sensors to collect vital signs and patient data in real time. By deploying deep neural networks, like transformer models, directly onto these devices, the system can locally analyze data, make immediate health assessments, or predict medical events without the need to send or store sensitive patient data in centralized cloud servers, thus enhancing user privacy and data security. This approach not only enhances the patient privacy by keeping their data within the local devices but also allows for faster, potentially life-saving decisions by reducing the latency associated with data being sent to the cloud and the cloud processing of the data. Moreover, it ensures continuous patient monitoring and real-time feedback even in the event of long-range network failures, making healthcare more resilient and accessible, especially in remote or underserved areas.

However, deploying transformer-based tools at the Edge presents a significant challenge for edge or IoT devices due to the intensive computational and memory requirements of transformer models. For instance, the Vicuna-13B chatbot [98] requires 26 GB of memory for the model parameters and substantial computational resources for inference. As discussed in Chapter 3, distribution methods[21, 58] for distributed DNN inference have been explored to bridge the gap between limited edge device resources and the demands of large DNN models. However, these methods generally assume continuous availability of all participating devices, which is often unrealistic due to potential device unavailability or failures.

Addressing this issue, our study emphasizes the need for robust partitioning methods for distributed transformer inference. Distributed inference across multiple devices offers a promising solution for handling large transformer models (e.g., Llama[99]) that exceed the memory capacities of individual devices, such as IoT devices, smart surveillance cameras, user laptops, etc. Existing frameworks, like Alpa [100] and DeepSpeed [101], effectively support distributed Large Language Model (LLM) training through data and model parallelism, but do not address at all robust distributed in-

ference on edge devices and do not cater for resource heterogeneity in edge systems or IoT settings.

Therefore, this chapter introduces a novel method, called **EASTER**, designed to learn robust distribution strategies for LLMs (transformers) that ensure functional inference and maintain close-to-original results under potential device failures. Learning such optimal strategies to distribute millions of neurons is challenging because a vast and complex design space needs to be explored. As shown in Figure 2.2, typical transformer-based models consist of several encoder and decoder blocks stacked together. The embedding dimension within each block, which represents the size of vectors used to encode images, words, or tokens, usually exceeds 1000. For example, if the embedded dimension of an encoder block is 768 [11], and we consider each dimension-related connection as a neuron, then the encoder block has 768 neurons. If we want to distribute these 768 neurons over four devices evenly, the exact number of possible distributions is $\binom{768}{192} \times \binom{576}{192} \times \binom{384}{192}$. The vast number of potential possibilities to distribute just one encoder block across multiple devices is almost unimaginable, let alone when considering the distribution of multiple blocks in large transformer models. There is a critical need to explore this extensive design space efficiently to identify a neuron distribution strategy that maintains performance against potential device failures to ensure the robustness and reliability of the distributed system.

To this end, we have developed a variant of the *Upper Confidence bounds applied to Trees* (UCT) algorithm [102] to efficiently narrow down the design space by considering the neuron importance in the transformer layers, enabling robust and memory-efficient splitting of transformer models across multiple devices. For different splitting strategies (design points) in the search space, our algorithm is designed to efficiently explore and identify optimal design points in the vast space, aiming to enhance splitting and prioritizing sub-spaces with the highest potential for robustness. It achieves this by adaptively and recursively splitting the design space into several sub-spaces and learning the expected rewards associated with different sub-spaces, effectively tackling the challenge posed by the extensive search space. By navigating and sampling the most and potential promising sub-spaces rather than the entire vast space, our approach enhances search efficiency, while balancing exploration and exploitation to avoid the pitfalls of local optima. The final Pareto points/solutions offer an optimal blend of robustness against device failures and operational efficiency regarding computation and memory.

We also automate the process of dividing transformer models for distributed computing by converting them into a unified neural network intermediate representation (IR). This step is followed by automated code generation and the subsequent deployment of the models across multiple edge devices. Our experimental results demonstrate that the system configurations identified as Pareto optimal points through the aforementioned design space exploration (DSE) method not only maintain system robustness but also achieve a notable reduction in memory usage. Furthermore, these configurations reduce the end-to-end inference latency for very large transformer models, demonstrating the effectiveness of our approach in optimizing both the performance and efficiency of distributed deep learning systems.

Our main novel contributions are summarized as follows:

- A novel UCT-based design space exploration algorithm is proposed that efficiently narrows down the vast design space, facilitating the discovery of effective model partitioning strategies for robust transformer distribution that balance performance and resource usage.
- By empirical validation, we demonstrate the efficacy of our EASTER method using typical transformers like ViT-16 [11], GPT2-Large [103], and Vicuna-7B [98], showcasing resilient model performance in image and common reasoning tasks.
- We provide the first implementation of an end-to-end tool for splitting transformer models, and also validate the advantages of distributed inference in terms of end-to-end inference latency and memory utilization compared to single-device inference.

6.2 RELATED WORK

The proliferation of transformer models in various applications has necessitated their adaptation beyond the confines of powerful cloud computing resources, directing significant research interest toward edge deployments. This section reviews pertinent literature across three main themes relevant to our work on EASTER: 1) adaptation of large transformer models for resource-constrained edge devices, 2) resilience against device failures, and 3) efficiency in design space exploration.

1) Adaptation of Transformer Models for Edge Constraints. The push towards deploying AI capabilities at the Edge, driven by privacy concerns, latency reduction, and energy efficiency, has seen approaches like model

compression [104–106] and neural architecture search [107–110] gain prominence. Such approaches can compress original transformer models to smaller models for resource-constrained devices. However, they typically require iterative retraining and may result in accuracy loss. Another approach is to deploy the original models onto distributed edge computing platforms such as health care systems [111], smart home systems [112], etc., in order to leverage all available resources collaboratively. Traditional layer and data partitioning methods like [21, 113] are applied to fully distribute the workload of a large Convolution Neural Network or a transformer-based model among multiple edge devices, thereby reducing the required computation resources of edge devices [23]. It involves breaking down a model's computational graph into smaller, manageable parts that can be processed in parallel across multiple devices. This is particularly challenging in edge computing due to the heterogeneous nature of devices and their limited computational capabilities. Model parallelism techniques like AlpaServe[100] developed for homogeneous data center clusters are targets for multi-batch inference which would perform poorly for single batches in heterogeneous edge environments. PipeEdge [113] partitions a neural network model into multiple pipeline stages and applies a dynamic programming (DP) algorithm to determine the optimal partition scheduling strategy for heterogeneous computation and communication. However, all of the aforementioned approaches and methods assume that the involved edge computing devices and communication links between them are always available and work properly. In contrast, our partitioning approach not only aims at maintaining computational efficiency but also considers the resilience of the system against possible temporary or permanent failures of devices, an aspect often overlooked in conventional partitioning strategies.

2) Resilience against Edge Failures. Resilience against device failures at the Edge concerns the property of a model being resilient in terms of inference accuracy to the failure of physical computing devices due to power outages, unstable inter-device connections, other hardware/software failures, etc. In distributed inference settings, the missing neurons mapped on those failed devices may result in a significant accuracy drop of CNN or transformer models (Figure 6.1(b)). Existing approaches and methods to mitigate this risk introduce various strategies. ElasticDL, introduced by Jun et al. [114], represents a significant advancement by integrating fault tolerance and elastic scheduling within a Kubernetes-native deep learning framework. While ElasticDL enhances system resilience and adaptability,

its practical deployment on edge devices is hampered by Kubernetes' complexity and the limited computational resources of edge environments.

Further contributions by Zhou et al. [58] and Li et al. [115] explore adaptive mechanisms to sustain operational continuity amidst device disruptions. Zhou et al. focus on adaptive computation offloading and dynamic resource allocation by proposing a versatile yet complex strategy for real-time adaptation. On the other hand, Li et al. propose the AR-MDI algorithm which facilitates resilient model distribution to minimize inference times on extensive datasets. However, these solutions often face challenges in scalability and efficiency, particularly when dealing with large models or a highly diverse network of edge devices, which may exacerbate coordination and data exchange overheads. DeepFogGuard [85] adopts a distinct approach by incorporating hyperconnections in the distributed inference process, enabling the system to bypass failed nodes. While this method demonstrates resilience to specific types of failures, such as inter-layer communication breakdowns or node failures within a multi-node inference pipeline, it remains limited to predetermined failure scenarios and requires extensive retraining to be implemented effectively.

In contrast to the aforementioned approaches, our method **EASTER** introduces a comprehensive solution designed to enhance the resilience of transformer models in the face of the unpredictable and dynamic nature of edge computing environments. Unlike previous methods that often rely on additional hardware resources, complex orchestration, or prior knowledge of potential failure types, **EASTER** employs a novel partitioning strategy that inherently accommodates multiple device failures without necessitating extra devices or computational redundancy. Our approach leverages advanced machine learning techniques to adaptively distribute model computations across edge devices, optimizing for both resilience and resource efficiency. By intelligently partitioning the model in a manner that anticipates and mitigates the impact of device failures, **EASTER** ensures robust inference accuracy under a wide range of failure conditions without the limitations imposed by specific assumptions or the need for supplementary computational overhead.

3) Efficiency in Design Space Exploration (DSE). In the context of Design Space Exploration (DSE), the original UCT algorithm [102], known for its efficacy in balancing the exploration-exploitation trade-off in single-objective optimization problems, is ingeniously adapted to the multi-objective optimization landscape in our work. This adaptation involves selecting promising parts of the search space by not only leveraging the UCT's inher-

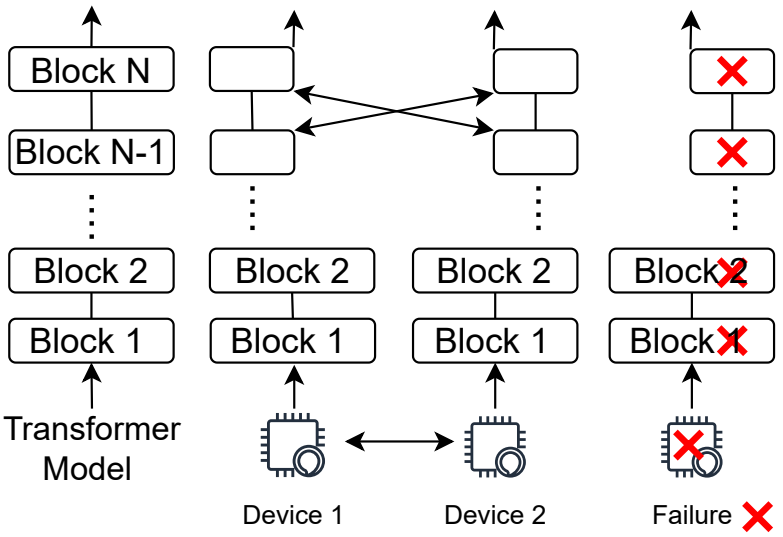
ent strengths but also enhancing it with traditional machine learning techniques for more efficient splitting and exploration of the design space. Such an integration significantly augments the UCT framework, enabling it to navigate complex, multi-dimensional optimization problems with greater precision and efficiency.

Existing DSE methods such as the Multi-Objective Tree-structured Parzen Estimator (MOTPE) [116] and the Non-dominated Sorting Genetic Algorithm II (NSGA-II discussed in Section 2.3.1) are well-known for their efficiency in multi-objective optimization. MOTPE is renowned for its sample efficiency and capability to handle high-dimensional spaces through its Bayesian optimization framework, which is particularly beneficial in scenarios with limited evaluation budgets. NSGA-II, on the other hand, excels in finding a diverse set of solutions across the Pareto front through its evolutionary algorithm, effectively managing the trade-offs between conflicting objectives. However, existing methods fall short in adapting to our specific scenario, which requires robust splitting of the transformer model block by block while simultaneously optimizing memory usage and inference latency. These methods lack customization for navigating the vast design space of our scenario.

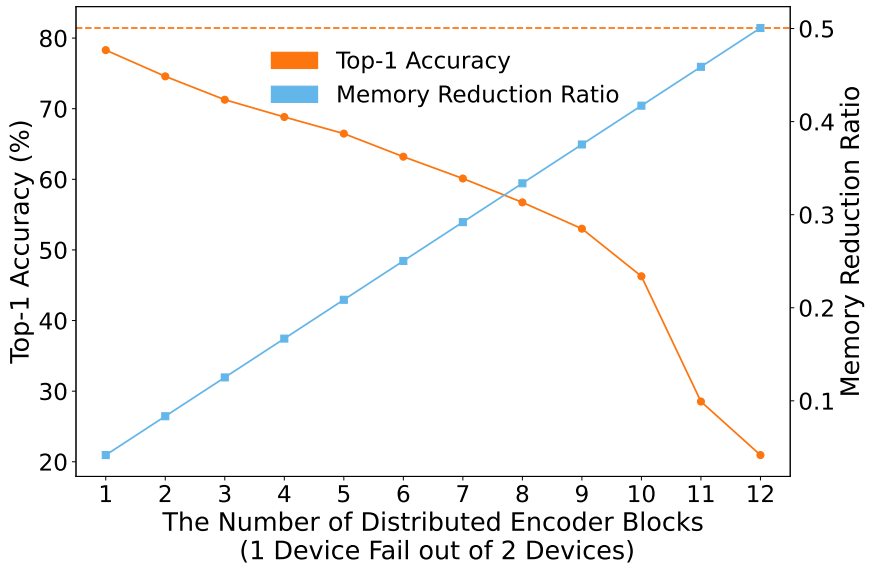
To address this gap, we enhance the UCT algorithm with machine learning techniques to combine the UCT's exploration-exploitation mechanism with the predictive and generalization capabilities of machine learning. This not only provides an efficient method to identify and explore promising spaces but also enhances the algorithm's ability to adaptively refine its search strategy based on learned insights. Our enhanced UCT approach, when compared to methods like MOTPE and NSGA-II, offers a complementary strategy ideally suited for scenarios where understanding and leveraging the structure of the search space is crucial. This tailored approach significantly boosts our search efficiency and the quality of outcomes, making it a particularly effective solution for our specific robustness needs for splitting transformer models.

6.3 ROBUST MODEL SPLITTING

In this section, we provide an example to illustrate why splitting a transformer model robustly is needed and why DSE matters in this context. Moreover, we describe how transformers can be splitted in a robust fashion.



(a) Layer Partitioning Method [21]



(b) The Top-1 Accuracy vs Memory Reduction Ratio

Figure 6.1: Comparative analysis of layer partitioning and its impact on memory reduction and accuracy.

6.3.1 *Motivational Example*

The process of splitting a transformer model for distributed inference across edge devices is crucial for running large models in environments with lim-

ited resources. Although some frameworks like PipeEdge [113] could distribute transformer models across multiple IoT devices using vertical partitioning and the accompanying orchestration, the crux of the problem lies in the robustness of the pipeline paradigm they utilize: a single failure within the pipeline can compromise the entire computation process. Thus, our discussion focuses on an alternative paradigm, namely partitioning the layers themselves within a deep learning model across multiple devices [21] (i.e., horizontal partitioning). A transformer model, composed of N encoder or decoder blocks, is designed for various tasks such as classification or text generation. As illustrated in Figure 6.1(a), by dividing blocks in the transformer model into two parts evenly, specifically on a block-by-block basis, we can distribute its workload across two devices. Each device then processes its allocated half blocks, necessitating periodic synchronization of their intermediate results to maintain consistency throughout the computation process. However, such a distribution strategy still introduces a vulnerability: should one of the two devices fail (as illustrated on the right-hand side of Figure 6.1(a)), it results in the loss of half the blocks' processing capability, thereby significantly impacting the model's overall performance and reliability. This scenario underlines the need for a robust distribution strategy that can minimize the risk and impact of device failures.

Taking the ViT-16 transformer model [11] as an example, it contains 12 encoder blocks stacked one by one. The significant impact of a device failure on the model performance is highlighted in Figure 6.1(b). When splitting and distributing the model's blocks across two devices, a device failure leads to a substantial drop in Top-1 accuracy, as critical block information is lost. This scenario is graphically represented with Top-1 accuracy (red line) and memory reduction ratio per device (blue line) against the number of distributed blocks (x-axis), demonstrating that as more blocks are distributed instead of fully replicated, the memory efficiency on the operational device improves, but at the cost of reduced accuracy due to the potential loss of computational resources during a device failure. For instance, when distributing all 12 encoder blocks of the ViT model across two devices, should one device fail due to a power outage or disconnection, half of the weights and intermediate results would be lost. In such a scenario, the Top-1 accuracy could drop to 20.95%, significantly impairing the model performance of distributed inference.

This trade-off between memory reduction per device and model accuracy underlines the challenge of finding a method to split encoder/decoder blocks that maximizes model accuracy retention while achieving optimal

memory efficiency. The goal is to develop a strategy that ensures even if one or more devices fail, the distributed model can maintain as much of its original performance as possible. As mentioned in Section 6.1, given the vast design space for distributing neurons in each encoder/decoder block, it is crucial to employ Design Space Exploration (DSE) to identify the most efficient distribution pattern, aiming to minimize accuracy loss while optimizing model deployment in distributed environments.

6.3.2 *Robust Model Splitting*

In the context of a transformer model containing N encoder or decoder blocks, we introduce an innovative uneven splitting method, called *Partial Split*, for distributing these blocks across multiple devices with robustness in mind. This method particularly aims at enhancing the model's resilience to device failures while reducing the memory usage on each device.

As illustrated in Figure 6.2(a) for example, evenly distributing a transformer block among four edge devices poses a significant risk, namely the model functionality is severely compromised, for example, when three out of these four devices fail or lose connection, as only a minimal fraction of attention connections remains operational for inference. To address this vulnerability, our method diverges from this conventional even splitting approach.

Instead, our method illustrated in Figure 6.2(b) employs a strategic replication of a certain fraction r of critical connections (the yellow box) across multiple devices, based on their weight importance. The remaining, less critical connections (the large green box), constituting a $(1-r)$ fraction, are then evenly distributed. This selective replication ensures that even in the event of multiple device failures, the most vital connections within each transformer block are retained, thereby preserving the model functionality and inference capabilities to a large extent. During runtime, the device initiating an inference request for image classification or text generation tasks loads both the replicated part (the yellow box) and its split part (the small green box) of the model. The other devices in the network load only their respective split parts. Notably, the replicated part remains unloaded on these devices (the dotted yellow boxes). This runtime loading strategy ensures that extra replicas are not redundantly loaded on other devices, thereby optimizing resource utilization and enhancing overall system efficiency.

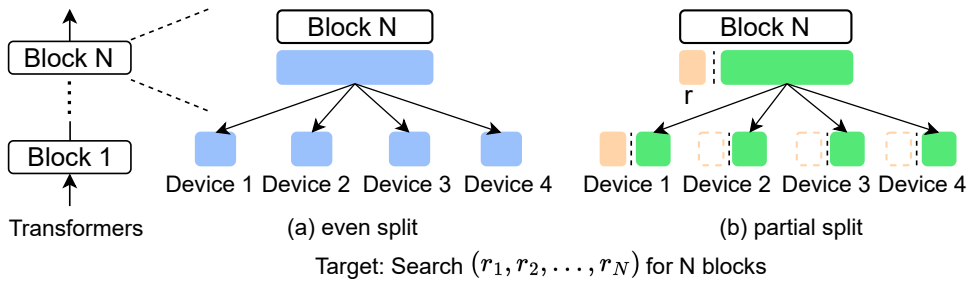


Figure 6.2: Partial Split

6.4 PROBLEM FORMULATION

The aforementioned uneven splitting method facilitates robust distribution of the computational workload of a transformer model across edge devices. However, the limited memory capacities of edge devices introduce challenges in determining the optimal fraction r for each transformer block that could preserve the model functionality and inference capabilities to a large extent. A large fraction r would require high memory usage per device, potentially exceeding the memory capacity of resource-limited edge devices. Conversely, a very small fraction r might compromise the proper model functionality in case multiple devices fail. Thus, an important trade-off emerges between the memory usage per device and the model functionality that is dependent on the fraction r of critical connections that are replicated for each block.

For a transformer model with N blocks, we define a parameter set $R = \{r_1, r_2, \ldots, r_N\}$, where $r_i \in [0..1]$ represents the fraction of replicated connections for block i. Each set of parameter values R corresponds to different memory usage m_j per device $D_j \in D$ and different model functionality in case some devices fail at runtime when a transformer model is distributed over a set of edge devices D. Therefore, our objective is to find an optimal set of parameter values R_{opt} which maximizes the model accuracy or performance score in case of failing devices with possible minimum memory usage $(m_1, m_2, \ldots, m_{|D|})$. Given the typically large value of N for prevalent transformer models and the continuous range of $r \in [0..1]$, a vast and complex design space needs to be explored in order to find an optimal solution.

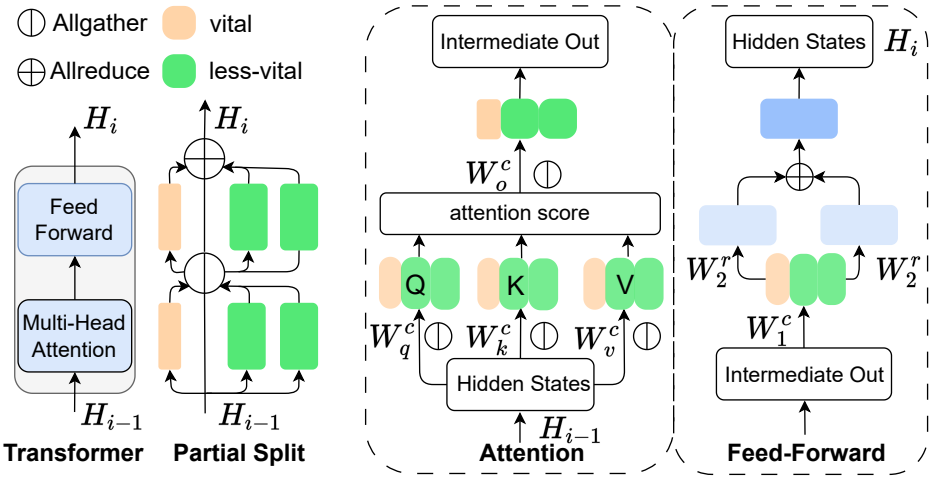


Figure 6.3: Transformer Partitioning

6.5 THE EASTER METHOD

In this section, we present our novel method designed to learn robust distribution strategies for transformer models against device failures that consider the trade-off between robustness (i.e., maintaining model functionality against failures) and resource utilization (including memory usage and computations). First, we provide more details about our robust partial split method introduced in Section 6.3. Next, we present our design space exploration (DSE) approach to solve the optimization problem, formulated in Section 6.4, that is required to achieve an efficient and robust partial split and distribution of transformer models on multiple edge devices. Finally, we introduce the end-to-end tool we have developed to automate our robust partial split method and distributed deployment of transformer models.

6.5.1 *Partial Split Method for Transformers*

In this section, we explain how the transformer model is split according to a parameter set R. Consider the example shown in Figure 6.3 where Block N in a transformer model is distributed across two devices and the obtained fraction $r_N \in R$ for this example is 0.25. The vital part of connections in the attention and feed-forward blocks is represented by the two yellow boxes that are both replicated across the two devices. The remaining, less-vital

part of connections for each block is split in two (the green boxes) and distributed evenly across the two devices.

To determine the vital part of connections, we calculate and use an importance score for each connection. For example, taking a general linear transformation in the feed-forward block, we first calculate the importance of connections corresponding to this linear transformation using the Taylor score [117] as follows:

$$I_{W^k} = |\Delta\mathcal{L}| = |\mathcal{L}_{W^k} - \mathcal{L}_{W^k=0}| \approx \left| \frac{\partial \mathcal{L}}{\partial W^k} W^k \right| \tag{1}$$

where I_{W^k} represents the importance score of the k_{th} connection/weights associated with the linear transformation, and $|\Delta\mathcal{L}|$ represents the loss changes when we remove this connection from the layer. After ranking the connections based on their importance score, we select the top $25\%(r = 0.25)$ of them to form the vital part of connections. Below, we provide details on how our partial split method is further tailored for the attention and feed-forward blocks within the transformer architecture to efficiently reduce computational workload and memory usage when the transformer is distributed across different edge devices.

6.5.1.1 *Attention Block*

As depicted in Figure 6.3, the hidden states H_{i-1} coming from the previous transformer block are transformed into queries (Q), keys (K), and values (V) using the weight matrices W_q, W_k, and W_v. Our method splits these matrices along their column dimensions (denoted by W_q^c, W_k^c, and W_v^c) and distributes them across devices. Consequently, each device generates the corresponding segments of Q, K, and V (denoted by the yellow and green boxes), necessitating an all-gather communication operation to concatenate the corresponding segments into complete Q, K, and V tensors. Taking the linear transformation with W_q weights (Figure 6.3) in the attention block as an example, the query matrix Q is generated by W_q. If the embedded dimension of the input tensor H_i is d_{model}, we compute the d_{model} importance scores for query Q using Equation 1. Once the replication factor r_i is determined, we rank and split the W_q weights along its column dimensions based on the rank indices derived from the d_{model} scores. We choose to replicate the top r% of weight W_q and allocate the remaining $(1-r)\%$ to multiple devices. For the small portion of W_q on each device, we replace the original matrix multiplication (matmul) operation with a small matmul operation containing its corresponding different part

of weight W_q. To maintain output accuracy, a communication operation for gathering the partial Q output is added after the small matmul. After the attention block multiplies the attention scores with values (V), the linear transformation with weight matrix W_o maps the multiplication result to match the dimension size of the intermediate output. In our method, we also split W_o into segments along the column dimension. Each segment of W_o^c produces a partial part of the intermediate output. Similarly, an extra all-gather communication operation is added to collect the segments and ensure the correctness.

Apart from these layers, the embedding layer follows a similar strategy for its matmul operation. However, we abstain from applying our partial split method to layernorm layers due to their relatively minimal weight and computational demand. Importantly, the full replication of layernorm weights on each device is prioritized to ensure model stability, given their significant role [118].

6.5.1.2 *Feed-Forward Block*

This block within a transformer block involves two linear transformations with weight matrices W_1 and W_2 to process the intermediate output and generate the hidden states H_i going to the subsequent transformer block. The first weight matrix is split along the column dimension (denoted as W_1^c in Figure 6.3). The second weight matrix is split along the row dimension (denoted as W_2^r). The partial output tensors (the yellow and green boxes) produced by W_1^c can directly go through the non-linear activation and serve as the input for the second linear transformation which is also split and denoted as W_2^r. This design eliminates the need for an all-gather operation to concatenate the partial outputs produced by the first linear transformation, thereby reducing both the computational workload per device and the inter-device communication overhead. Finally, a collective all-reduce operation is applied to sum the partial output from all devices to form the correct hidden states output H_i.

6.5.2 *Design Space Exploration*

To solve the optimization problem formulated in Section 6.4, we have devised a DSE approach that effectively navigates in the vast and complex design space mentioned in Section 6.4. Our DSE approach leverages supervised learning techniques to progressively concentrate the search for an optimal solution within increasingly smaller and more promising spaces,

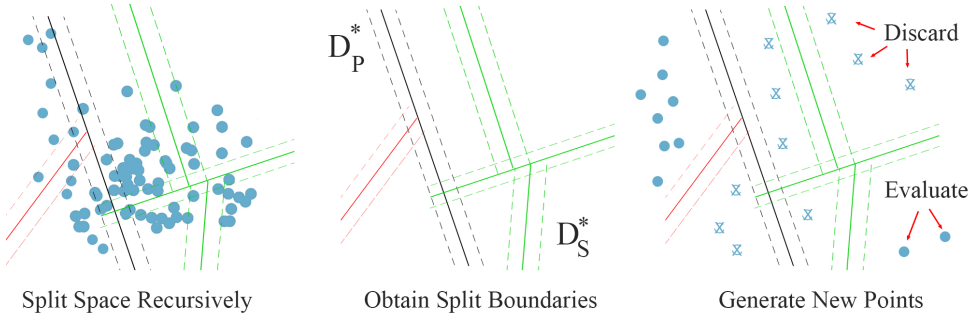


Figure 6.4: Our DSE approach

thereby enhancing search efficiency. As depicted in Figure 6.4, the approach starts by randomly generating several design points $\mathbb{R} = \{R_1, R_2, \cdots, R_p\}$ (blue points), and evaluate the objectives $\mathbb{F}(\mathbb{R})$ using the fitness function $\mathbb{F}$ for each design point $R_i \in \mathbb{R}$ to form an initial learnable space $D = (\mathbb{R}, \mathbb{F}(\mathbb{R}))$. Here, $R_i = \{r_1^i, r_2^i, \ldots, r_N^i\}$ is a set of fractions corresponding to a specific partial split strategy for all N blocks in a transformer model. The fitness function $\mathbb{F}$ concerns various conflicting objectives such as memory usage, performance, etc. $\mathbb{F}$ can be implemented by users, including analytical models, real measurements of distributed inference on multiple devices (as is done in this chapter), etc. Then, our DSE approach recursively splits the design space D and obtains a set of split boundaries. Subsequently, we apply these learned boundaries to generate new design points within specific promising design spaces to improve the search efficiency.

We apply the Upper Confidence Bound value [102] to identify which area within $\mathbb{R}$ is most likely to contain optimal design points and then concentrate our search on this smaller, promising area, denoted as D_P^* and shown in the middle of Figure 6.4. However, an early decision about the promising area might inadvertently overlook other areas that could contain optimal points as well. To mitigate this, while the majority of our design points are generated within the currently perceived promising area D_P^*, we also allocate a smaller portion of design points to generate from other spaces, represented by D_S^*. This approach iteratively learns the entire space $\mathbb{R}$ and allows us to more accurately identify the most promising regions for optimal points.

Algorithm 2 describes, in more detail, the aforementioned DSE approach illustrated in Figure 6.4. The algorithm consists of two main steps and takes as an input the maximum search trials Tr_{max}, the number of new random design points n_p for updating the search space D, a lower bound

Algorithm 2: Design Space Exploration

Input : Maximum trials Tr_{max}; Population size n_p; lower bound lb, exploration factor α;
Output: Space D_P with Pareto points;

Initialize randomly D with points (R, F_R): $D \leftarrow \{(R_1, \text{FITNESS}(R_1)), \cdots, (R_{n_p}, \text{FITNESS}(R_{n_p}))\}$
while $|D| \leqslant Tr_{max}$ **do**
  // Step 1: Narrow Down Search Space
  foreach $(R_i, F_{R_i}) \in D$ **do**
    if F_{R_i} is nondominated **then**
      $D_P \leftarrow D_P \cup (R_i, F_{R_i})$
  $D_S \leftarrow D \setminus D_P$; $\mathbb{CL}_P \leftarrow \emptyset$; $\mathbb{CL}_S \leftarrow \emptyset$
  $D_P^*, \mathbb{CL}_P = \texttt{NarrowDown}(D_P, \mathbb{CL}_P)$
  $D_S^*, \mathbb{CL}_S = \texttt{NarrowDown}(D_S, \mathbb{CL}_S)$
  // Step 2: Add New Random Points,
  // Evaluate and Update D
  $\mathbb{R}_P = \texttt{NewPoints}((1-\alpha) * n_p, \mathbb{CL}_P)$
  $\mathbb{R}_S = \texttt{NewPoints}(\alpha * n_p, \mathbb{CL}_S)$
  foreach $R_i \in (\mathbb{R}_P \cup \mathbb{R}_S)$ **do**
    $D \leftarrow D \cup (R_i, \text{FITNESS}(R_i))$
return D_P
Function NarrowDown(D, $\mathbb{CL}$):
  foreach $(R_i, F_{R_i}) \in D$ **do**
    $\mathbb{R}_D \leftarrow \mathbb{R}_D \cup R_i$
  $(\mathbb{R}_{D_1}, \mathbb{R}_{D_2}) = \text{KMeansTwoClustersOn}(\mathbb{R}_D)$
  $D_L = (\mathbb{R}_{D_1}, 1) \cup (\mathbb{R}_{D_2}, -1)$
  $CL, D_1, D_2 = \text{SVMTrainedOn}(D_L)$
  if $(|D| < lb) \vee (CL(D_1) = CL(D_2))$ **then**
    return D, $\mathbb{CL}$
  else
    $\text{UCB}(\mathbb{R}_{D_i}) = \overline{\mathbb{F}(\mathbb{R}_{D_i})} + \alpha\sqrt{\frac{\log|\mathbb{R}_D|}{|\mathbb{R}_{D_i}|}} : i = 1, 2$
    $\mathbb{R}_{D^*} = \underset{\mathbb{R}_{D_i}}{\text{argmax}}\ \text{UCB}(\mathbb{R}_{D_i})$; $D^* = (\mathbb{R}_{D^*}, \mathbb{F}(\mathbb{R}_{D^*}))$
    $\mathbb{CL} \leftarrow \mathbb{CL} \cup CL$
    return NarrowDown(D^*, $\mathbb{CL}$)
Function NewPoints(N, $\mathbb{CL}$):
  $\mathbb{R} \leftarrow \emptyset$
  while $|\mathbb{R}| < N$ **do**
    R = RandomPoint;
    $\mathbb{R} \leftarrow \mathbb{R} \cup R$
    foreach $CL_i \in \mathbb{CL}$ **do**
      if $CL_i(R) = -1$ **then**
        $\mathbb{R} \leftarrow \mathbb{R} \setminus R$
        break
  return $\mathbb{R}$

(lb) to determine the maximum number of design points in an unsplittable area, and the exploration factor α which determines the degree of exploration. A higher value for α encourages more exploration in the search space. The output of Algorithm 2 is space $D_P = \{(R_1, F_{R_1}), ..., (R_{|P|}, F_{R_{|P|}})\}$ of

Pareto-optimal solutions where every solution $R_i = \{r_1^i, r_2^i, \ldots, r_N^i\}$ is a set of fractions corresponding to a Pareto-optimal partial split strategy for all N blocks in a transformer model.

In line 2, we first randomly initialize a number of design points and evaluate their objectives using the fitness function, yielding an initial learnable search space D.

In Step 1 (lines 4-9), the algorithm narrows down the space via Support Vector Machine (SVM) classifiers and generates a series of SVM boundaries. In lines 4-7, we select the non-dominated points from D to create a new primary space marked as D_P, and the rest of the points are put into a new secondary space marked as D_S. In lines 8-9, the *NarrowDown* function is applied to recursively split D_P and D_S into smaller spaces D_P^* and D_S^*. Concurrently, all involved splitting SVM boundaries are aggregated into the boundary sets $\mathbb{CL}_P$ and $\mathbb{CL}_S$.

In Step 2 (lines 10-13), we generate new design points and evaluate these new design points using the fitness function FITNESS. To balance the exploration-exploitation trade-off, 80% of these new points ($\mathbb{R}_P$) are derived from D_P^* in line 10, while the remaining 20% (i.e., for $\alpha = 0.2$) of the new design points ($\mathbb{R}_S$) are derived from D_S^* in line 11. This ratio, while adjustable, typically requires experimental trials for better search efficiency. Then, we apply the fitness function to evaluate the objective values for these new points and add them to the search space D in line 13. This iterative process is repeated until the maximum number of trials T is reached (see line 3). Ultimately, the Pareto-optimal points comprising space D_P found by this DSE process represent the optimal solutions that balance the memory usage and the model functionality.

In lines 15-27, the *NarrowDown* function recursively splits the search space D and obtains a series of learned split boundaries $\mathbb{CL}$. In line 18, we initially employ the K-means clustering method to categorize/divide the design points within $\mathbb{R}_D$ into two distinct clusters $\mathbb{R}_{D_1}, \mathbb{R}_{D_2}$. Following this clustering, we calculate the average objective values for each cluster. The cluster with the higher average objective values is considered to be situated in a more favorable space. Consequently, in line 19, we assign a label of 1 to the design points in this more promising cluster, while design points in the less favorable cluster are labeled as -1, and we put all labeled points in a new set D_L. In line 20, we train the SVM classifier CL with the new set of labeled points D_L and split D_L into two spaces D_1 and D_2. In lines 21-22, if the number of design points in D is below the lower bound lb or if the SVM classifier CL predicts only a single category, both

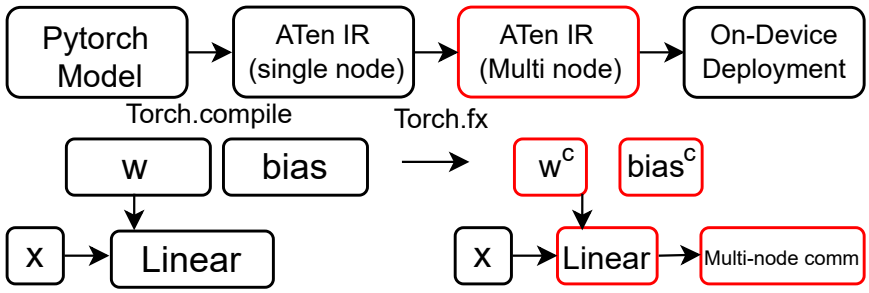


Figure 6.5: Multi-node IR conversion tool

indicating that space D is non-divisible, the recursive function *NarrowDown* terminates and returns the set of classifiers $\mathbb{CL}$. Otherwise, in lines 24-27, we mark the space with the larger UCB value [102], calculated in line 24, as the more promising design space D^*, and add the SVM classifier CL into the recursive splitting set $\mathbb{CL}$.

In lines 28-37, the *NewPoints* function randomly generates N new design points using the input set of SVM classifiers $\mathbb{CL}$. In line 31, a random design point R is generated and added to the set of new points $\mathbb{R}$. Then, point R is classified using the set of trained SVM classifiers $\mathbb{CL}$ in lines 33-36. That is, if all SVMs in $\mathbb{CL}$ classify point R to belong to the class with label 1 then point R remains in the set, otherwise it is removed (line 35). Finally, in line 37 the new set of random points $\mathbb{R}$ is returned.

6.5.3 *Multi-node Intermediate Representation*

We have developed an end-to-end tool that facilitates automated model partitioning and its distributed deployment, in line with one of the Pareto-optimal partial split strategies $R_i \in D_P$ found by our DSE Algorithm 2 presented in Section 6.5.2. In general, traditional frameworks for deep learning (DL) model deployment on edge devices, such as TVM [119], IREE [120], and others, do not sufficiently support distributed inference. Therefore, our end-to-end tool is specifically implemented to transform Convolutional Neural Networks or transformer models from Huggingface [121] into optimized multi-node computation graphs, thereby making them suitable for efficient deployment across multiple devices. Our tool is versatile enough to support both CNNs and transformer models but in this chapter we focus on its application to transformer models.

As illustrated in Figure 6.5, our tool begins by utilizing the existing 'torch.compile' method [122] to convert an initial PyTorch transformer model into the low-level ATen Intermediate Representation (IR) for a single node. Subsequently, an automated conversion process is employed to replace the single-node ATen IR into a multi-node variant. For instance, in handling linear transformations, the tool splits the associated coefficients and redefines new Linear transformations that are adapted to the altered shapes of coefficients or inputs as illustrated by the red boxes in Figure 6.5. Modifications to these operations are facilitated using 'torch.fx' [123], accommodating the new coefficient dimensions. Our own customized multi-node communication operations such as GatherByIndex, AllReduceByIndex, AllConcatByIndex, etc., are integrated after the modified operation (see red box "Linear" in Figure 6.5) to ensure the calculation correctness. To enhance the tool's versatility, we implement these communication operations in C++ such that they can be integrated into other inference engines. We have also developed a compatible interface that enables the conversion of this multi-node IR into formats supported by various other inference engines (e.g., NCNN [50], IREE, etc.). Its compatibility and ease of integration with these existing edge frameworks enhance both usability and scalability.

6.6 EVALUATION OF THE EASTER METHOD

In this section, we evaluate our EASTER method empirically to demonstrate its efficacy on typical transformer models and showcase resilient models' performance. We describe our experimental setup followed by presenting and discussing some experimental results obtained during automated DSE experiments, we have performed using Algorithm 2 and the end-to-end tool introduced in Section 6.5.3.

6.6.1 *Experimental Setup*

To evaluate EASTER, we perform experiments with three typical transformer models, namely ViT-16 [11], GPT2-Large [103], and Vicuna-7B [98] representing three different kinds of transformer architectures, taken from the Huggingface open-source community [121]. Given their widespread use in image and text tasks, and their diversity in transformer blocks, operation counts, and memory requirements, we consider these transformers to be representative targets to demonstrate the merits of our method. We compare the searching efficiency of our Algorithm 2 on these models with two

state-of-the-art multi-objective optimization algorithms, namely the NSGA-II Genetic Algorithm [5] and MOTPE [116]. The task of our DSE experiments is to simultaneously minimize the maximum memory usage per device and the model performance score (loss) under severe device failures. To ensure a fair comparison with NSGA-II and MOTPE, we set the maximum number of search iterations to 1000 for each DSE experiment. The search time for the three methods are quite similar, with most of the time being consumed by the objective evaluations. For the first objective (maximum memory usage per device), we assess the peak memory usage during the program runtime. We normalize its value range to $[0, 1]$ by dividing the memory usage $m_j(R_i)$ by the total memory usage on a single device D_j. Lower values indicate reduced replication and more balanced model distribution. To evaluate the second objective (performance score S) of the models, we employ distinct techniques tailored to each model's specific domain. For the ViT-16 model, we measure the Top-1 error score on the ImageNet-1k dataset for image tasks. A lower error represents higher image classification capabilities, and the lower the error the better. For the two Large Language Models (LLMs) - GPT2-Large and Vicuna-7B - we utilize zero-shot perplexity (PPL) analysis on the WikiText2 and PTB datasets to assess the models' language understanding and generalization capabilities. A lower PPL score, especially in a zero-shot context, means a better ability to handle unseen data.

To validate the performance of Pareto-optimal points from the DSE process using Algorithm 2, we apply the split fractions R_i, found by the algorithm, to the two LLMs by distributing each LLM across four devices, i.e., four GPU units in our experiments. We disable three GPU units to simulate severe device failure scenarios in order to assess the models' robustness. We apply a separate and more diverse collection of reasoning and generative datasets [124] to test the models' performance (robustness) against severe failures in practical reasoning tasks, namely ARC-easy, ARC-challenge, WinoGrande, HellaSwag, BoolQ, PIQA, and OpenbookQA. These diverse datasets provide a comprehensive platform for testing the models' reasoning and generative capabilities.

To evaluate the resilience of our methods under varying failure conditions, we deployed three models across four edge devices and examined model performance in scenarios where 1 (**1D-Fail**), 2 (**2D-Fail**), or 3 devices (**3D-Fail**) experience failures. We take the state-of-art layer partitioning method (**LP**) [21] from the domain of distributed CNN inference as the inspiration to implement a similar method for linear operations within

encoder/decoder blocks of transformer models, which does not entail importance consideration. Subsequently, we benchmark our approach against this LP-based partitioning method for transformer models in terms of robustness.

For the three transformer models, we assess the robustness of our method using different sets of R values for the partial split strategy, allowing for a comprehensive comparison of how well each method retains model performance against device failures.

We use the experimental test-bed described in Section 2.3.3.2 to test distributed inference for transformers. We demonstrate the functionality of our multi-node implementation, generated by our end-to-end tool introduced in Section 6.5.3, and the advantages of distributing large transformer models over multiple edge devices/boards by conducting a series of benchmarks on the aforementioned edge test-bed using the three representative transformer models ViT-16, GPT2-Large, and Vicuna-7B under four different distributed system configurations: single device, two devices, four devices, and eight devices. In all experiments, transformer blocks were evenly distributed across the devices. We mainly evaluate two metrics: overall end-to-end inference latency and memory reduction with different distribution configurations.

The end-to-end latency (T) of a model is measured from the time a user input is received until the time the complete output is generated. For the ViT-16 model, user inputs are images with dimensions (3x224x224), whereas for the two LLMs (GPT2-Large and Vicuna-7B), user inputs are sequences of 128 tokens. The reported latency is computed by averaging time T for 100 user inputs. To measure T and break it down to computation time (T_{cal}) and communication/synchronization overhead (T_{comm}) in our distributed inference execution, we employ a specific adjustment of the timeout parameter values in our multi-node communication operations introduced in Section 6.5.3. More specifically, setting the timeout values to zero permits each device to function independently, i.e., without inter-device data communication and synchronization delays, thereby enabling the measurement of the pure computation time T_{cal}. Altering the timeout values to one second activates inter-device communication and synchronization actions besides the pure computations, thereby facilitating the measurement of the total end-to-end inference latency T. We then determine the communication/synchronizaton overhead T_{comm} by calculating the difference $T - T_{cal}$, thereby effectively quantifying the additional time needed for inter-device data communication and synchronization.

Table 6.1: Zero-shot performance (max. per-device memory usage and accuracy-%) with three out of four edge devices failing

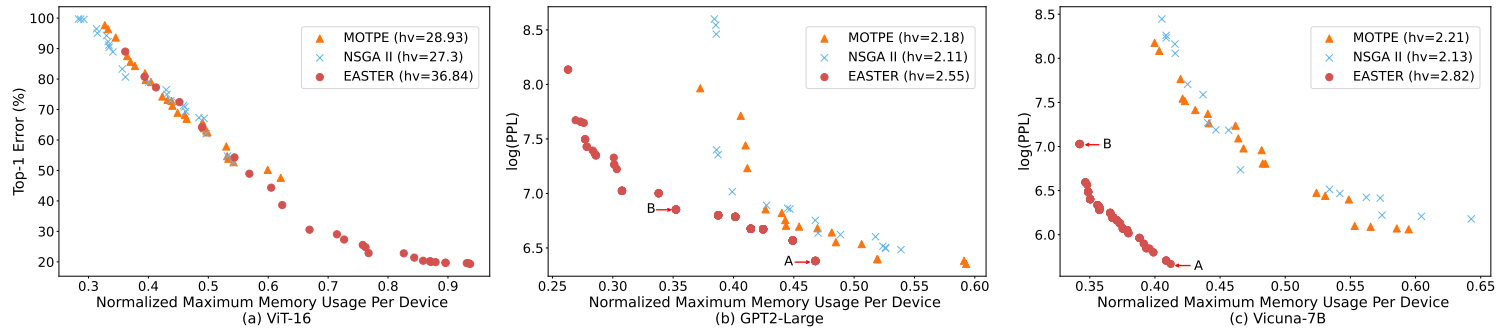
Models	Memory (reduction ratio)	ARC-c	ARC-e	WinoGrande	HellaSwag	OBQA	PIQA	BoolQ
Vicuna-7B (R=1)	27.00 GB(baseline)	43.17	75.63	69.46	56.48	33.00	77.31	80.98
Vicuna-7B (A)	11.27 GB(-58.25%)	28.41	46.00	56.51	34.58	20.60	63.93	62.26
Vicuna-7B (B)	9.18 GB(-66.00%)	20.90	29.63	52.09	28.32	16.00	57.07	45.17
GPT2-Large (R=1)	3.20 GB/(baseline)	21.67	53.16	55.33	36.40	19.40	70.35	60.49
GPT2-Large (A)	1.50 GB/(-53%)	17.92	31.02	49.64	26.67	13.40	57.78	54.22
GPT2-Large (B)	1.12 GB/(-65%)	17.75	30.18	50.36	26.47	13.80	55.93	54.22

To determine the aforementioned memory reduction, we continuously monitor the peak memory usage of each device in our edge test-bed during runtime for every distributed system configuration.

6.6.2 *DSE Results and Comparison*

We have performed three distinct DSE experiments for the ViT-16, GPT2-Large, and Vicuna-7B models by employing our EASTER method and Algorithm 2 along with the NSGA-II and MOTPE algorithms for comparison purposes. The Pareto-optimal points found by each of these three algorithms are separately plotted in Figure 6.6. The yellow triangles represent the points found by MOTPE, the blue crosses represent NSGA-II points, and the red dots correspond to points found by our Algorithm 2 within EASTER. The x-axis in Figure 6.6(a), (b), and (c) represents the normalized maximum memory usage per device explained in Section 6.6.1. The y-axis represents the Top-1 error for ViT-16 and the PPL for GPT2-Large and Vivcuna-7B. The rationale behind using the Top-1 error and PPL is explained in Section 6.6.1.

To quantitatively assess the effectiveness of EASTER, NSGA-II, and MOTPE, as well as to compare them, we calculate the well-known and widely-used hypervolume metric (hv), based on the Pareto-optimal points plotted in Figure 6.6, that serves as an indicator of the search space coverage in DSE. As shown in Figure 6.6, our EASTER method and algorithm demonstrate superior performance because of the higher hypervolume value hv, indicating more effective search space coverage of EASTER compared to NSGA-II and MOTPE. For example, the Pareto-optimal points found by EASTER for Vicuna-7B and shown in Figure 6.6(c) dominate those found by NSGA-II and MOTPE, resulting in higher hypervolume value of 2.82 and highlighting the EASTER effectiveness in identifying optimal solutions.



(a) ViT-16 (b) GPT2-Large (c) Vicuna-7B

Figure 6.6: Comparison of DSE results delivered by EASTER, NSGA-II, and MOTPE for ViT-16, GPT2-Large, and Vicuna-7B

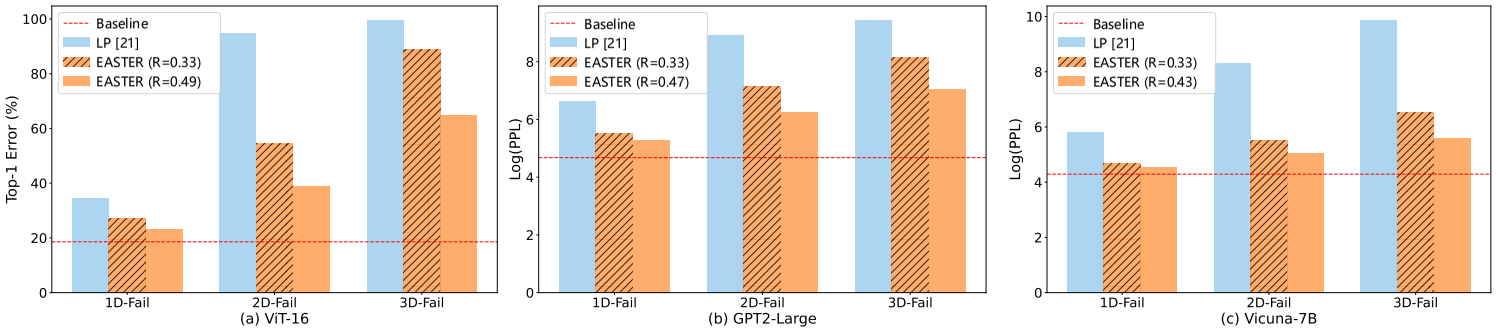


Figure 6.7: Robustness comparison of EASTER with Layer-wise Partitioning across four devices

As explained in Section 6.6.1, we apply the split fractions R_i, found by Algorithm 2, to the models by distributing each model across four devices. Moreover, we disable *three of the four* devices in order to simulate severe device failure scenarios to assess the models' robustness. The results for the LLMs (GPT2-Large and Vicuna-7B) are shown in Table 6.1.

The first column specifies two different R_i settings, named A and B, for each of the two LLMs together with the baseline setting, named R=1. The baseline setting R=1 for each LLM is the original model fully replicated over the four devices with no loss of model weights/connections due to failures. Note that the evaluation metrics associated with settings A and B are also shown in Figure 6.6(b) and (c) - see the red dots marked with A and B. The second column in Table 6.1 shows the maximum memory usage per device under the aforementioned settings. The remaining columns show the evaluation accuracy (in %) of the operational part of the model, i.e. the part still running on the non-failing device, across several zero-shot open-ended tasks on widely recognized common sense reasoning datasets [124]: ARC-e(asy), ARC-c(hallenge), WinoGrande, HellaSwag, BoolQ, PIQA, and OpenBookQA.

Analyzing the results in the second column of Table 6.1, we observe that the memory reduction for settings A and B compared to the baseline setting clearly confirms the efficacy of our EASTER method. For example, the Vicuna-7B model experiences a significant memory reduction of up to 66% (from 27.00 GB to 9.18 GB), although this comes with certain accuracy trade-offs across the evaluated tasks. Here, task ARC-c shows the highest accuracy sensitivity to memory reduction, i.e., a decrease of 46% in accuracy. The GPT2-Large model in setting B with a memory reduction of 65% shows a relatively minor performance decline in terms of accuracy for datasets like WinoGrande and BoolQ. This resilience, especially when the memory is decreased from 3.20 GB to 1.12 GB, suggests a difference in inherent robustness between GPT2-Large and Vicuna-7B, with the former displaying greater robustness. However, it is important to note that our DSE method and algorithm prioritize the optimization for general PPL scores, rather than tailoring the search to enhance specific task scores. Overall, both models demonstrate a notable degree of performance resilience under extreme failure scenarios, indicating their potential for effective deployment in environments with memory constraints, such as edge devices.

6.6.3 *Robustness Verification Against Varying Failures*

To deepen our understanding of EASTER's robustness, we performed a comparative study against the conventional layer-wise partitioning approach, specifically under scenarios involving various numbers of device failures. To maintain a fair comparison, we designated the transformer model to be distributed across four devices in both methodologies.

In the context of the layer-wise (LP) partitioning method, our experimental setup involved randomly failing devices for the 1D-Fail, 2D-Fail, and 3D-Fail scenarios, subsequently averaging the model's performance to represent its expected behavior across different failure conditions. This approach allowed us to obtain the average performance under diverse device failure circumstances for the LP method. For our EASTER method, we similarly averaged the model's performance across the same failure scenarios, allowing for a balanced and insightful comparison.

As depicted in Figure 6.7, the x-axis categorizes the failure scenarios (1D-Fail, 2D-Fail, or 3D-Fail), whereas the y-axis quantifies model performance, measured by the Top-1 accuracy on the ImageNet-1k validation dataset or perplexity (PPL) value. Please note the logarithmic scale for the PPL scores. The graphical representation uses blue bars to indicate the performance of the traditional layer-wise partitioning (LP) method in the face of device failures, while orange bars illustrate the performance of our EASTER method. To assess the impact of redundancy, we examine two distinct R values, analyzing their influence on device failure resilience.

For instance, in a 2D-fail scenario, the Top-1 error of the LP method reaches 94.742%, in contrast to our method, which significantly lowers the error rate to 54.626%. By adjusting the R value from 0.33 to 0.49, we observe a further reduction in the Top-1 error to 38.792%. Similarly, with the Vicuna7B model, the logarithmic value of perplexity (PPL) observed using the LP method under a 2D-Fail condition is 8.29. In contrast, our method achieves a $\log(\text{PPL})$ of 5.50 with an R value of 0.33. Further refining the R value to 0.43 results in an even lower $\log(\text{PPL})$ of 5.05.

These results clearly demonstrate that our EASTER method significantly outperforms the LP method in maintaining model performance against device failures. Moreover, increasing the R value, which dictates the degree of neuron replication, can further improve model robustness.

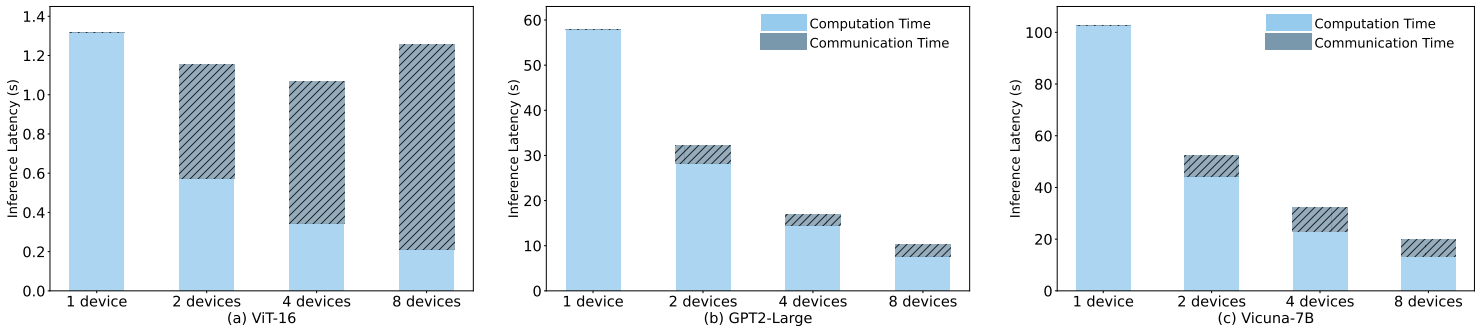


Figure 6.8: Inference Latency and Communication Time for Different Models Across Device Configurations

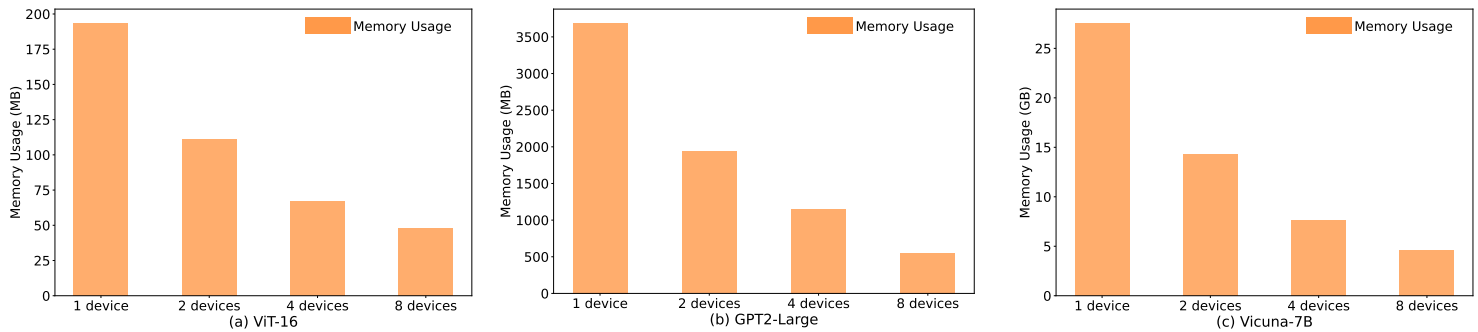


Figure 6.9: Memory Usage for Different Models Across Device Configurations

6.6.4 *Distributed Inference*

In this section, we evaluate our end-to-end tool that facilitates automated model partitioning and its deployment on distributed edge devices. Our tool is specifically implemented to convert standard PyTorch transformer models into optimized multi-node implementations following our EASTER method, making the models suitable for efficient distributed deployment on edge devices. We present empirical results, obtained by using our edge testbed described in Section 2.3.3.2, in order to demonstrate the advantages of EASTER in terms of overall end-to-end inference latency and maximum memory usage per device in a distributed system running transformer models. Here, in all experiments, transformer blocks are evenly distributed across the devices. In Figure 6.8, the light blue bars represent the computation time T_{cal} of the distributed inference process, the grey blue bars indicate the communication/synchronization overhead T_{comm}, whereas the orange bars in Figure 6.9 denote the maximum memory usage per device. The data is presented for different numbers of collaborating edge devices across the three models.

As shown in Figure 6.8, in most cases, the overall end-to-end inference latency improves when increasing the number of edge devices. As the number of devices increases, in all cases, computation time T_{cal} (light blue bars) reduces correspondingly. Only in the case of ViT-16 (Figure 6.8(a)), this advantage is counterbalanced by a rise in the communication overhead (gray bars), which, in an eight-device setup, surpasses the computational savings, leading to an overall increase in the inference latency. Conversely, for GPT2-Large, the communication overhead, while increasing with more devices, still remains a smaller fraction compared to the computation time. This results in a near-linear acceleration, with an overall inference latency decrease from 58.00 seconds using one device to 7.62 seconds using eight devices. The increase in communication overhead therefore seems more pronounced in smaller transformer models like ViT-16, that represents a fundamental trade-off between computation and communication.

The results shown in Figure 6.9 clearly indicate that with an increasing number of devices (from 1 to 8 devices), there also is a noticeable decrease in memory usage per device. For instance, the maximum on-device memory usage for ViT-16 decreases from 193.8 MB in a single-device configuration to 48.1 MB in an eight-device configuration. Similarly, GPT2-Large exhibits a significant memory reduction from 3.6 GB on a single device to 556.3 MB across eight devices. A significant reduction in memory usage per

device from 27.6 GB on a single-device configuration to 4.6 GB on an eight-device configuration is observed for Vicuna-7B as shown in Figure 6.8(c). Such reduction enables the models to run the complete float32 version at the Edge without the need for extra swap space or model quantization, highlighting EASTER's effectiveness in memory savings per device.

Finally, if the reduction in computation time due to distributed inference is outweighed by the increase in communication time, the overall end-to-end latency increases. To manage this, we apply the timeout mechanism discussed in Section 5.4.1 within the distributed inference implementation, allowing us to adjust time thresholds to ensure that the communication overhead does not overshadow the computational benefits. If communication delays exceed the set timeout, the system is configured to continue processing without completing the communication. This approach helps prevent the communication overhead from becoming excessively burdensome.

The above findings validate the efficiency of EASTER in optimizing memory usage per device in distributed transformer inference, particularly in edge computing environments where resource constraints are a critical factor.

6.7 CONCLUSIONS

This chapter introduces EASTER, a novel method designed to robustly partition transformer models across edge devices, effectively addressing the challenge of potential device failures at the Edge. The EASTER method navigates the vast design space of splitting strategies by learning the expectations of different design sub-spaces. It also outperforms traditional state-of-the-art DSE methods in searching efficiency for our distribution problem. Through extensive experimentation, EASTER has been proven to identify Pareto solutions within a limited number of experimental trials efficiently.

Utilizing our developed end-to-end tool, we have the capability to evaluate the distributed implementation on actual hardware boards, which allows us to confirm the advantages in memory usage and inference latency that distributed inference brings. Moreover, our findings substantiate that partial splitting significantly enhances model robustness in the face of device failures. This approach not only minimizes memory consumption on each device but also has the potential to reduce overall end-to-end latency,

presenting a valuable opportunity for deploying large-scale transformer models within edge computing environments.

7

CONCLUSION

This PhD thesis investigates the efficient and robust distributed deployment of deep neural network (DNN) models on resource-constrained edge devices. As DNNs, particularly large language models, continue to grow in complexity and size, they trigger an escalating demand for resources such as computation power, memory, and energy. When deployed at the Edge, these resources are often severely limited due to the nature of edge devices, that are typically limited hardware with limited battery life.

Despite these constraints, deploying DNNs on edge devices offers several advantages over cloud-based services. Centralized cloud servers pose significant privacy risks and often cannot meet the low-latency requirements of time-sensitive applications. By deploying DNNs closer to the data source — at the Edge — these issues can be substantially mitigated.

To deploy DNNs at the Edge, model compression techniques like pruning and quantization are applied to reduce the size of DNNs, making them more suitable for the limited computational capabilities of edge devices. However, such compression can sometimes impact the accuracy of the models and require extensive retraining iterations. An alternative strategy for deploying DNNs at the Edge involves leveraging all available resources across multiple devices. This approach entails partitioning a large DNN model and distributing the partitions to run on separate edge devices. Implementing this strategy requires efforts to appropriately split and manage the DNN partitions across different devices, ensuring that each partition functions correctly and efficiently in collaboration with others. In addition, this approach maintains the accuracy of the original DNN and utilizes the collective computing resources of multiple devices at the Edge, thereby resolving limitations posed by single-device deployments.

In this thesis, we explore and address the challenges (Chapter 1) associated with distributed DNN inference, dividing the study into two parts. The first part of the thesis focuses on the distribution strategies of DNNs across multiple edge devices. This part involves the development of sys-

tems and tools designed to enable the rapid and efficient deployment of DNNs for distributed inference. It also includes exploring optimal configuration solutions for distributing DNNs across multiple devices, to optimize memory usage, energy consumption, and throughput. The second part of the thesis is dedicated to enhancing the robustness of the distributed inference system, particularly its resilience against device failures or unstable connections. This involves creating fault-tolerant mechanisms that maintain the functionality of the distributed DNN execution and ensure the correctness of inference results, even in the face of device failures or connectivity disruptions. The main contributions of this thesis, in terms of overcoming the challenges, are summarized below:

- AutoDiCE: a framework designed for the distributed deployment of CNN models across resource-constrained edge devices. It automates the partitioning of CNN models into sub-models, facilitates the generation of C++ code for their distributed execution, and ensures efficient communication across multiple resource-constrained edge devices via the MPI protocol.

- Advanced Design Space Exploration (DSE) Method: this DSE method incorporates a novel genetic encoding technique to explore efficient distribution strategies of CNN models. It aims to enhance system inference throughput while minimizing the energy and memory requirements on each device. In addition, it employs a multi-stage hierarchical DSE process by leveraging both analytical models and real measurements through the AutoDiCE framework to search for optimal or near-optimal distribution solutions efficiently.

- RobustDiCE: this method improves the robustness of distributed DNN inference by grouping and distributing neurons evenly across multiple devices according to their neuron importance scores. It ensures that the DNNs continue functioning effectively, even if some devices become temporarily unavailable, thus maintaining consistent system performance.

- EASTER: a method that involves a novel DSE process to determine a series of neuron replication ratios within each layer of DNNs, particularly in large transformer models. It strategically balances robustness against device failures with resource utilization and performance goals, facilitating efficient and effective distributed inference.

7.1 ANSWERS TO CHALLENGES

In this section, we reflect on the challenges set forth in Chapter 1 and discuss how this thesis addressed them.

CHL1: *How to flexibly and efficiently offload DNN models over multiple edge devices?*

To address this challenge, we developed AutoDiCE, the first fully automated framework for distributing CNN models across multiple resource-constrained devices at the Edge. This framework seamlessly manages the entire workflow, from splitting a trained CNN model based on a CNN partitioning specification to deploying it on multiple edge devices. To this end, it automates the splitting of a CNN model into a set of sub-models and code generation for distributed and collaborative execution of these sub-models.

At the core of AutoDiCE is the model splitting process, where its user interface interprets the pre-trained CNN model using the ONNX format for compatibility with major deep learning frameworks like PyTorch and TensorFlow. Following this, the model is divided into several sub-models according to the provided mapping specifications. Subsequently, the back-end of AutoDiCE undertakes the generation of optimized C++ code for each sub-model tailored to the specific hardware configurations of the edge devices involved. This includes using MPI (Message Passing Interface) for data communication and synchronization among the sub-models, crucial for exploiting parallelism within and among edge devices. Moreover, AutoDiCE generates necessary configuration files and setups such as MPI rank files to ensure that each sub-model operates synchronously and efficiently across the distributed network. In the final phase, AutoDiCE packages the generated code, configuration files, and sub-models into tailored deployment packages for each device. These packages contain all the necessary components to execute the sub-models as independent MPI processes on the edge devices.

Further details of AutoDiCE are discussed in Chapter 3, where a comprehensive example illustrates all steps in action. This example describes the platform specification, which specifies the computational and software resources available on the edge devices. Additionally, it outlines a mapping specification that details how each CNN layer is distributed across the specified hardware resources, using JSON format for clarity. This in-action exam-

ple provides insights into the operational steps of AutoDiCE and showcases its practical application in distributing CNN models for edge computing.

Using three representative CNN models from the ONNX model zoo, AutoDiCE has been thoroughly tested to demonstrate its capabilities. The outcomes from the distributed execution of these models on multiple devices have shown substantial reductions in energy consumption per device and optimized memory usage compared to single-device configurations. Overall, AutoDiCE offers a sophisticated solution to the first challenge that dramatically simplifies and automates the process of deploying distributed CNN inference across edge devices, addressing key challenges such as efficient model splitting, automated code generation, and optimal resource utilization.

CHL2: *How to perform an efficient DSE process to find optimal distribution strategies of DNN models at the edge while considering multiple optimization objectives?*

To determine the optimal distribution strategy for DNN models at the Edge, in Chapter 4 we introduced a multi-stage hierarchical Design Space Exploration (DSE) method, utilizing a custom-tailored NSGA-II method as the search engine. Our DSE method begins with a novel chromosome encoding method called Split Point Encoding (SPE). This method reduces the large search space by partitioning the CNN model into N groups of consecutive layers so that only consecutive CNN layers are mapped onto a single processing element, minimizing unnecessary data communication between processing elements. Furthermore, the multi-stage design in our DSE process enhances the convergence of the DSE process by retaining information from the Pareto points of DNN distributions from previous stages. For instance, inspired by the SPE encoding method, Pareto distribution solutions for two devices may include a split point that can be utilized for dividing DNNs for four devices. By leveraging Pareto-optimal solutions from earlier stages as the basis for subsequent levels of DSE, we accelerate our search by focusing on specific parts of the design space, building on the best solutions identified in earlier stages.

To further accelerate the DSE process, we also employ a two-level hierarchical search within each stage. The first level uses analytical models within an NSGA-II engine to approximate each objective function — such

as throughput, memory, and energy consumption — thus avoiding time-consuming real on-device evaluations. Near-optimal solutions from this level, combined with Pareto-optimal solutions from a previous stage, serve as the first-generation parents for the second-level DSE. In this second level, real measurements from AutoDiCE-generated CNN inference implementations are utilized to establish the Pareto front for the subsequent DSE stage. The outcomes of the final DSE stage provide the definitive Pareto-optimal solutions.

In Chapter 4, we rigorously evaluated our multi-stage hierarchical DSE method through extensive experiments. The results reveal that traditional encoding methods frequently get stuck in dominance-resistant solutions, failing to discover high-quality mappings even after extensive iterations. In contrast, our SPE encoding method significantly enhances both the efficiency and quality of the search, achieving superior mappings within just 20 hours of search time.

By incorporating multiple stages and hierarchical layers into the DSE process, we further improve the discovery of high-quality mappings. For example, after 40 hours of search time, our SPE-based DSE outperforms the performance of a basic DSE approach across all objectives. This demonstrates the substantial advantages of our method in effectively navigating the complex design space of CNN distributions for edge deployments, providing more efficient and optimal mappings compared to the conventional DSE method.

CHL3 : *How to ensure the robustness of DNN inference across multiple edge devices against possible failures or transient unavailability?*

To address this challenge, we have developed a method named RobustDiCE, detailed in Chapter 5, aimed at robust distributed DNN inference at the Edge. RobustDiCE strategically evaluates the importance of each neuron within DNN layers to assess their criticality to ensure maintained inference accuracy in scenarios of potential device failures. By employing a combination of partial neuron replication and importance-aware neuron clustering, RobustDiCE distributes neurons across multiple devices in an even and robust way. This distribution not only balances the computational load but also ensures that critical neurons remain functional, thus enhancing both system and model robustness.

The core strategy, the partial replication of essential neurons, mitigates the impact of device failures. This strategy ensures that the remaining devices can continue operating important neurons, allowing the system to maintain an inference accuracy level comparable to full-replication where no neurons are lost, but with substantially lower computational resource demands.

We tested our novel partitioning method using the ImageNet-1K dataset across several representative CNN models under pessimistic device failure scenarios. We compare with the state-of-the-art methods and demonstrate the effectiveness of RobustDiCE in delivering efficient and robust distributed CNN inference against device failures. Our method not only optimizes resource utilization (memory usage, energy consumption per device, and system throughput) but also ensures comparable inference accuracy against failures.

CHL4 : *How to determine an optimal set of neuron replication ratios for each layer within a distributed DNN model that maximizes inference accuracy in the face of potential edge device failures, while minimizing the impact on memory usage and computational load per device, thereby ensuring efficiency in terms of energy consumption and overall system performance?*

To address our fourth research challenge, we introduced EASTER, a novel Design Space Exploration (DSE) method outlined in Chapter 6. EASTER is designed to determine optimal replication ratios for DNN layers, based on the assumption that minor adjustments to these ratios minimally impact the final objective values of fitness functions. This method splits the design space into smaller, manageable sub-spaces based on these objective values, thereby largely accelerating the DSE process.

For different distribution strategies (design points) of transformer models, which involve a vast design space, our algorithm is designed to efficiently and quickly explore and identify optimal design points, enabling robust and memory-efficient splitting of transformer models across multiple devices. We first efficiently narrow down the design space by considering the neuron importance in the transformer layers, as this assessment allows us to group neurons within each layer, significantly reducing their distribution complexity. Further, we achieve this by adaptively and recursively splitting the design space into several sub-spaces and learning the expected

rewards associated with different sub-spaces, effectively tackling the challenge posed by the extensive search space. We have developed a variant of the *Upper Confidence bounds applied to Trees* (UCT) algorithm [102], aiming to enhance splitting and prioritizing sub-spaces with the highest potential for robustness. By recursively navigating and sampling the most potentially promising sub-spaces rather than the entire vast space, our approach enhances search efficiency, while balancing exploration and exploitation to avoid the pitfalls of local optima. The final Pareto points/solutions offer an optimal blend of robustness against device failures and operational efficiency regarding computation and memory.

Our evaluation with three transformer models (ViT, GPT-2, and Vicuna) demonstrates EASTER's ability to reduce memory usage and improve end-to-end latency for inference while maintaining model accuracy and performance, especially under device failure conditions. Additionally, we automate the process of distributing transformer models by converting them into a unified neural network intermediate representation (IR), followed by automated code generation and deployment with the AutoDiCE framework. Our end-to-end implementation enables the evaluation of transformer models on actual hardware, making it an effective strategy for deploying large-scale transformer models in user applications at the Edge.

7.2 FUTURE WORK

Given the discussions in and contributions of this thesis, several areas could be fruitful for future research work to extend the methodologies and insights developed in the thesis:

1. **Cross-Device Optimization:** A promising topic for future research involves developing advanced algorithms to optimize network traffic and data flow between devices in distributed networks, aiming to minimize latency and maximize throughput. This optimization might include using reinforcement learning to select and quantize specific layers of neural networks, reducing the on-device computations and communications between devices. By distributing these partially compressed layers across multiple computing devices, communication overhead can be decreased due to quantization. However, this selective compression strategy requires careful balance to optimize data flow without sacrificing model accuracy. The exploration process includes determining which layers to compress and to what extent, to maintain the model's accuracy and meet latency requirements. While compress-

ing more layers can reduce communication overhead and computational demands, it could also adversely affect the model's accuracy. Thus, a critical challenge is to quantify the trade-offs between reducing latency and preserving model performance integrity.

2. **Management of Distributed Systems:** There is a significant opportunity to exploit scalable runtime deployment strategies that dynamically adjust to the fluctuating number and capabilities of edge devices in heterogeneous environments. This research could also explore methods for automatic recovery and reconfiguration of neural network tasks in response to hardware failures or unexpected device downtimes. Developing systems that can adapt to changing network conditions and device availabilities without human intervention could dramatically increase the resilience and flexibility of distributed neural networks.

3. **Security and Privacy:** As distributed neural networks often operate in environments where edge devices process sensitive data, enhancing the security and privacy of these systems is crucial. Future research could focus on developing robust encryption methods and privacy-preserving algorithms that protect data without compromising the operational efficiency of the neural network. For instance, distributing DNNs increases the number of potential entry points for attackers. Considering several smartphones and IoT devices are performing real-time analytics, each device, if compromised or attacked, could allow unauthorized access to the entire network. This vulnerability is exacerbated by the frequent data synchronization among devices, where each transmission can potentially be intercepted or manipulated. Consequently, there is a pressing need for research into advanced cryptographic techniques and intrusion detection systems that can secure these data exchanges without degrading the network's performance.

These future research directions not only aim to address the existing challenges but also push the boundaries of what distributed neural networks can achieve, making them more efficient, secure, and resilient in dynamic and diverse environments.

BIBLIOGRAPHY

[1] Nestor Maslej et al. *Artificial Intelligence Index Report 2023*. 2023. arXiv: 2310.03715 [cs.AI].

[2] Waqas Tariq Toor, Maira Alvi, and Mamta Agiwal. "Combined access barring scheme for IoT devices using Bayesian estimation." In: *Electronics* 9.12 (2020), p. 2191.

[3] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[4] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[6] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." In: (2014). arXiv: 1409.1556 [cs.CV].

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: (2015). arXiv: 1512.03385 [cs.CV].

[9] 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." In: (2017). arXiv: 1704.04861 [cs.CV].

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[11] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." In: *ICLR*. 2020.

[12] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. "Gpt-4 technical report." In: (2023). arXiv: 2303.08774 [cs.CL].

[13] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks." In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[14] Yiping Kang et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge." In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 615–629.

[15] 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)*. IEEE. 2017, pp. 328–339.

[16] En Li, Zhi Zhou, and Xu Chen. "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy." In: *Proceedings of the 2018 workshop on mobile edge communications*. 2018, pp. 31–36.

[17] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. "Modnn: Local distributed mobile computing system for deep neural network." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1396–1401.

[18] Zhuoran Zhao et al. "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters." en. In: *IEEE TCAD* 37.11 (Nov. 2018), pp. 2348–2359. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2018.2858384. (Visited on 04/20/2021).

[19] 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*. Springer, 2019, pp. 77–90.

[20] Ramyad Hadidi, Jiashen Cao, Michael S. Ryoo, and Hyesoon Kim. "Toward Collaborative Inferencing of Deep Neural Networks on Internet-of-Things Devices." en. In: *IEEE Internet of Things Journal* 7.6 (June 2020), pp. 4950–4960. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2020.2972000. (Visited on 11/15/2021).

[21] Rafael Stahl et al. "DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices." In: *International Journal of Parallel Programming* 49.4 (2021), pp. 600–624.

[22] Erqian Tang and Todor Stefanov. "Low-Memory and High-Performance CNN Inference on Distributed Systems at the Edge." In: *Proc. of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC)*. ACM, 2021, pp. 1–8.

[23] Xiaotian Guo, Andy D Pimentel, and Todor Stefanov. "Automated Exploration and Implementation of Distributed CNN Inference at the Edge." In: *IEEE IoT Journal* 10.7 (2023).

[24] Andy D. Pimentel. "Methodologies for Design Space Exploration." In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay. Singapore: Springer Nature Singapore, 2022, pp. 1–31. ISBN: 978-981-15-6401-7. DOI: 10.1007/978-981-15-6401-7_23-1. URL: https://doi.org/10.1007/978-981-15-6401-7_23-1.

[25] Carlo R Raquel and Prospero C Naval Jr. "An effective use of crowding distance in multiobjective particle swarm optimization." In: *Proceedings of the 7th Annual conference on Genetic and Evolutionary Computation*. 2005, pp. 257–264.

[26] SN Sivanandam, SN Deepa, SN Sivanandam, and SN Deepa. *Genetic algorithms*. Springer, 2008.

[27] *NVIDIA Jetson Xavier NX*. 2020. URL: https://developer.nvidia.com/embedded/jetson-xavier-nx.

[28] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[29] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[30] Microsoft. *ONNX Runtime*. 2018. URL: https://github.com/microsoft/onnxruntime.

[31] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

[32] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. "Automated Exploration and Implementation of Distributed CNN Inference at the Edge." In: *IEEE Internet of Things Journal* 10.7 (2023), pp. 5843–5858. DOI: 10.1109/JIOT.2023.3237572.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *nature* 521.7553 (2015), pp. 436–444.

[34] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and Sundaraja S Iyengar. "A survey on deep learning: Algorithms, techniques, and applications." In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[36] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. "Recent advances in deep learning for speech research at Microsoft." In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 8604–8608.

[37] Tharam Dillon, Chen Wu, and Elizabeth Chang. "Cloud Computing: Issues and Challenges." In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 2010, pp. 27–33. DOI: 10.1109/AINA.2010.187.

[38] Kanil Patel, Kilian Rambach, Tristan Visentin, Daniel Rusev, Michael Pfeiffer, and Bin Yang. "Deep Learning-based Object Classification on Automotive Radar Spectra." In: *2019 IEEE Radar Conference (RadarConf)*. 2019, pp. 1–6. DOI: 10.1109/RADAR.2019.8835775.

[39] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer. 2015, pp. 234–241.

[40] Russell Reed. "Pruning algorithms-a survey." In: *IEEE transactions on Neural Networks* 4.5 (1993), pp. 740–747.

[41] Yunhui Guo. "A survey on methods and theories of quantized neural networks." In: (2018). URL: https://arxiv.org/abs/1808.04752.

[42] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network." In: *arXiv preprint arXiv:1503.02531* (2015).

[43] AutoDiCE. *https://github.com/parrotsky/AutoDiCE*. 2022.

[44] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. 2019. URL: https://github.com/onnx/onnx.

[45] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. "CoAtNet: Marrying Convolution and Attention for All Data Sizes." In: (2021). arXiv: 2106.04803 [cs.CV].

[46] Yanping Huang et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." In: *Advances in neural information processing systems* 32 (2019).

[47] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. "PipeDream: Generalized Pipeline Parallelism for DNN Training." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, 1–15. ISBN: 9781450368735. DOI: 10.1145/3341301.3359646.

[48] Wei Yang Bryan Lim et al. "Federated Learning in Mobile Edge Networks: A Comprehensive Survey." In: *IEEE Communications Surveys Tutorials* 22.3 (2020), pp. 2031–2063. DOI: 10.1109/COMST.2020.2986024.

[49] Xuefei Yin, Yanming Zhu, and Jiankun Hu. "A Comprehensive Survey of Privacy-Preserving Federated Learning: A Taxonomy, Review, and Future Directions." In: *ACM Comput. Surv.* 54.6 (2021). ISSN: 0360-0300. DOI: 10.1145/3460427.

[50] Lihui Tencent. *NCNN*. 2017. URL: https://github.com/Tencent/ncnn.

[51] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. 2013–2016. URL: http://pjreddie.com/darknet/.

[52] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *International Conference on Learning Representations*. 2015.

[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: (2015). arXiv: 1512.03385 [cs.CV].

[54] Gao Huang, Zhuang Liu, Geoff Pleiss, Laurens Van Der Maaten, and Kilian Weinberger. "Convolutional Networks with Dense Connectivity." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[55] ONNX. *ONNX model zoo*. 2022. URL: https://github.com/onnx/models.

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[57] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. "Hierarchical design space exploration for distributed CNN inference at the edge." In: *3rd Workshop on IoT, Edge and Mobile for Embedded Machine Learning (ITEM 2022), part of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2022, pp. 545–556. DOI: 10.1007/978-3-031-23618-1_36.

[58] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. "Adaptive parallel execution of deep neural networks on heterogeneous edge devices." In: *SEC*. 2019, pp. 195–208.

[59] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices." In: *IEEE/ACM Transactions on Networking* 29.2 (2020), pp. 595–608.

[60] Xueyu Hou, Yongjie Guan, Tao Han, and Ning Zhang. "DistrEdge: Speeding up convolutional neural network inference on distributed edge devices." In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2022, pp. 1097–1107.

[61] Mohammad Loni, Sima Sinaei, Ali Zoljodi, Masoud Daneshtalab, and Mikael Sjödin. "DeepMaker: A multi-objective optimization framework for deep neural networks in embedded systems." In: *Microprocessors and Microsystems* 73 (2020), p. 102989.

[62] Svetlana Minakova, Dolly Sapra, Todor Stefanov, and Andy D Pimentel. "Scenario Based Run-Time Switching for Adaptive CNN-Based Applications at the Edge." In: *ACM Transactions on Embedded Computing Systems (TECS)* 21.2 (2022), pp. 1–33.

[63] Lie Meng Pang, Hisao Ishibuchi, and Ke Shang. "NSGA-II with simple modification works well on a wide variety of many-objective problems." In: *IEEE Access* 8 (2020).

[64] *RobustDiCE Software implementation*. 2022. URL: https://anonymous.4open.science/r/RobustDiCE-4C09.

[65] Xiaotian Guo, Quan Jiang, Andy D. Pimentel, and Todor Stefanov. "Model and System Robustness in Distributed CNN Inference at the Edge." In: *Integration, the VLSI Journal (Submitted)*.

[66] Jia Deng et al. "ImageNet: A large-scale hierarchical image database." In: *2009 IEEE Conference on CVPR*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[67] Zonghao Hou et al. "Design and implementation of heartbeat in multi-machine environment." In: *17th International Conference on Advanced Information Networking and Applications, 2003. AINA 2003.* IEEE. 2003, pp. 583–586.

[68] Ramyad Hadidi, Jiashen Cao, Bahar Asgari, and Hyesoon Kim. "Creating Robust Deep Neural Networks With Coded Distributed Computing for IoT." In: *IEEE International Conference on Edge Computing and Communications*. 2023.

[69] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. "Modnn: Local distributed mobile computing system for deep neural network." In: *IEEE DATE*. 2017, pp. 1396–1401.

[70] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. "A systematic review on software robustness assessment." In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–65.

[71] Walfredo Cirne, Francisco Brasileiro, Daniel Paranhos, Luís Fabrício W Góes, and William Voorsluys. "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems." In: *Parallel Computing* 33.3 (2007), pp. 213–234.

[72] John Paul Walters and Vipin Chaudhary. "Replication-based fault tolerance for MPI applications." In: *IEEE Transactions on Parallel and Distributed Systems* 20.7 (2008), pp. 997–1010.

[73] Harumasa Tada, Makoto Imase, and Masayuki Murata. "On the robustness of the soft state for task scheduling in large-scale distributed computing environment." In: *2008 International Multiconference on Computer Science and Information Technology*. IEEE. 2008, pp. 475–480.

[74] Shashank Rajput et al. "DETOX: A redundancy-based framework for faster and more robust gradient aggregation." In: *Advances in Neural Information Processing Systems* 32 (2019).

[75] Nicholas Cheney, Martin Schrimpf, and Gabriel Kreiman. "On the robustness of convolutional neural networks to internal architecture and weight perturbations." In: *arXiv preprint arXiv:1703.08245* (2017).

[76] Kunping Huang, Paul H Siegel, and Anxiao Jiang. "Functional error correction for robust neural networks." In: *IEEE Journal on Selected Areas in Information Theory* 1.1 (2020), pp. 267–276.

[77] Vinay Amatya, Abhinav Vishnu, Charles Siegel, and Jeff Daily. "What does fault tolerant deep learning need from mpi?" In: *Proceedings of the 24th European MPI Users' Group Meeting*. 2017, pp. 1–11.

[78] Cheng Liu et al. "Fault-Tolerant Deep Learning: A Hierarchical Perspective." In: (2022). URL: https://arxiv.org/abs/2204.01942.

[79] Cesar Torres-Huitzil and Bernard Girau. "Fault and error tolerance in neural networks: A review." In: *IEEE Access* 5 (2017), pp. 17322–17341.

[80] Zeinab Hakimi. "Collaborative Inference for Distributed Camera System." MA thesis. The Pennsylvania State University, 2019.

[81] Sohei Itahara, Takayuki Nishio, and Koji Yamamoto. "Packet-loss-tolerant split inference for delay-sensitive deep learning in lossy wireless networks." In: *IEEE GLOBECOM*. 2021, pp. 1–6.

[82] Ashkan Yousefpour et al. "Resilinet: Failure-resilient inference in distributed neural networks." In: *arXiv preprint arXiv:2002.07386* (2020).

[83] Jani Boutellier, Bo Tan, and Jari Nurmi. "Fault-Tolerant Collaborative Inference through the Edge-PRUNE Framework." In: (2022). URL: https://arxiv.org/abs/2206.08152.

[84] Xin He et al. "AxTrain: Hardware-oriented neural network training for approximate inference." In: *Proceedings of the international symposium on low power electronics and design*. 2018, pp. 1–6.

[85] Ashkan Yousefpour et al. "Guardians of the deep fog: Failure-resilient DNN inference from edge to cloud." In: *Workshop on challenges in artificial intelligence and machine learning for IoT*. 2019, pp. 25–31.

[86] Jose Luis Bernier, Julio Ortega, E Ros, Ignacio Rojas, and Alberto Prieto. "A quantitative study of fault tolerance, noise immunity, and generalization ability of MLPs." In: *Neural Computation* 12.12 (2000), pp. 2941–2964.

[87] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. "Pruning filters for efficient convnets." In: *arXiv* (2016). URL: https://arxiv.org/abs/1608.08710.

[88] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. "Soft filter pruning for accelerating deep convolutional neural networks." In: (2018). arXiv: 1808.06866 [cs.CV].

[89] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. "Snip: Single-shot network pruning based on connection sensitivity." In: (2018). arXiv: 1810.02340 [cs.CV].

[90] Seul-Ki Yeom et al. "Pruning by explaining: A novel criterion for deep neural network pruning." In: *Pattern Recognition* 115 (2021), p. 107899.

[91] Bent Fuglede and Flemming Topsoe. "Jensen-Shannon divergence and Hilbert space embedding." In: *Int. symposium on Information theory*. 2004, p. 31.

[92] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Comm. of the ACM* 60.6 (2017), pp. 84–90.

[93] Zhuang Liu et al. "A ConvNet for the 2020s." In: *CVPR* (2022).

[94] Xiaotian Guo, Quan Jiang, Yixian Shen, Andy D. Pimentel, and Todor Stefanov. "EASTER: learning to split transformers robustly at the Edge." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2024). DOI: 10.1109/TCAD.2024.3438995.

[95] Yihan Cao et al. "A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt." In: *arXiv:2303.04226* (2023).

[96] OpenAI. "GPT-4 Technical Report." In: (2023). arXiv: 2303.08774 [cs.CV].

[97] Mahantesh N Birje and Savita S Hanji. "Internet of things based distributed healthcare systems: a review." In: *Journal of Data, Information and Management* 2 (2020), pp. 149–165.

[98] Lianmin Zheng et al. *Judging LLM-as-a-judge with MT-Bench and Chatbot Arena*. 2023.

[99] Hugo Touvron et al. "Llama 2: Open foundation and fine-tuned chat models." In: *arXiv preprint arXiv:2307.09288* (2023).

[100] Zhuohan Li et al. "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving." In: *OSDI 23)*. 2023, pp. 663–679.

[101] Jeff Rasley et al. "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters." In: *Proceedings of the 26th ACM SIGKDD*. 2020, pp. 3505–3506.

[102] Levente Kocsis et al. "Bandit based monte-carlo planning." In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[103] Alec Radford et al. "Language models are unsupervised multitask learners." In: *OpenAI blog* 1.8 (2019), p. 9.

[104] Mohammad Wali Ur Rahman et al. "Quantized transformer language model implementations on edge devices." In: *arXiv:2310.03971* (2023).

[105] Yelysei Bondarenko et al. "Understanding and overcoming the challenges of efficient transformer quantization." In: *arXiv:2109.12948* (2021).

[106] Zhikai Li et al. "I-ViT: integer-only quantization for efficient vision transformer inference." In: *ICCV*. 2023, pp. 17065–17075.

[107] Chengyue Gong et al. "Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training." In: *ICLR*. 2021.

[108] Krishna Teja Chitty-Venkata et al. "Neural architecture search for transformers: A survey." In: *IEEE Access* 10 (2022), pp. 108374–108412.

[109] Yong Guo et al. "Nat: Neural architecture transformer for accurate and compact architectures." In: *NeurIPS* 32 (2019).

[110] Sheng Li et al. "Hyperscale Hardware Optimized Neural Architecture Search." In: *ASPLOS*. 2023, pp. 343–358.

[111] Sita Rani et al. "IoT equipped intelligent distributed framework for smart healthcare systems." In: *Towards the Integration of IoT, Cloud and Big Data: Services, Applications and Standards*. Springer, 2023, pp. 97–114.

[112] Biljana L Risteska Stojkoska et al. "A review of Internet of Things for smart home: Challenges and solutions." In: *Journal of cleaner production* 140 (2017), pp. 1454–1464.

[113] Yang Hu et al. "Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices." In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2022, pp. 298–307.

[114] Jun Zhou et al. "ElasticDL: A Kubernetes-native Deep Learning Framework with Fault-tolerance and Elastic Scheduling." In: *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*. 2023, pp. 1148–1151.

[115] Pengzhen Li et al. "Adaptive and Resilient Model-Distributed Inference in Edge Computing Systems." In: *IEEE Open Journal of the Communications Society* (2023).

[116] Yoshihiko Ozaki et al. "Multiobjective Tree-Structured Parzen Estimator." In: *Journal of Artificial Intelligence Research* 73 (2022), pp. 1209–1250.

[117] Xinyin Ma et al. "LLM-Pruner: On the Structural Pruning of Large Language Models." In: *NIPS* (2023).

[118] Ruibin Xiong et al. "On layer normalization in the transformer architecture." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10524–10533.

[119] Tianqi Chen et al. "TVM: An automated End-to-End optimizing compiler for deep learning." In: *OSDI 18*. 2018, pp. 578–594.

[120] Vanik Ben et al. *IREE: An MLIR-based compiler and runtime for ML models from multiple frameworks*. 2019. URL: https://iree.dev/.

[121] Thomas Wolf et al. "Huggingface's transformers: State-of-the-art natural language processing." In: *arXiv preprint arXiv:1910.03771* (2019).

[122] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation." In: *ASPLOS* (2024).

[123] James Reed et al. "torch. fx: Practical program capture and transformation for deep learning in python." In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 638–651.

[124] Leo Gao et al. *A framework for few-shot language model evaluation*. Version v0.0.1. Sept. 2021. DOI: 10.5281/zenodo.5371628.

SUMMARY

As deep neural networks (DNNs), especially large language models, become increasingly complex and huge, their computational demands escalate. DNN-based applications, typically are provided as services on cloud servers equipped with numerous GPUs. Due to the rapid expansion of Internet of Things (IoT) networks, characterized by their numerous connected devices, this development has necessitated a paradigm shift towards processing data closer to the source rather than in centralized cloud systems. It fosters a keen interest in deploying deep learning models at the edge. The term "edge" here refers to a variety of networked devices with computing capacity placed anywhere along the path of data transmission between a data source and the cloud.

Deploying DNNs at the edge offers enhanced privacy, security, and reliability, rendering this approach attractive yet challenging for user applications. The main challenge arises from the intensive computational demands of neural networks while edge devices usually have limited available resources. This discrepancy necessitates innovative solutions such as model compression, distillation, etc. to effectively run complex DNN models within the constraints of edge environments. However, these methods may introduce iterative retraining costs for specific user applications and potential accuracy drops in certain image or natural language processing tasks.

This thesis focuses on the collaborative execution of large DNNs across multiple edge devices, investigating strategies for efficient and robust model deployment on resource-constrained edge devices. The thesis is divided into two main parts:

The first part focuses on solving the challenges raised by the limited computational resources of edge devices. It involves developing systems and tools that facilitate the rapid and efficient distributed deployment of DNN models across multiple edge devices. We design the AutoDiCE framework for automating the distribution process. This framework simplifies the partitioning of models, automates code generation for distributed execution, and optimizes inter-device communication. Additionally, we also explore

optimal distribution configurations of DNN models for different objectives. An advanced Design Space Exploration (DSE) technique, which employs novel genetic encoding, efficiently searches for optimal distribution strategies of CNN models to minimize energy and memory usage while maximizing system throughput.

Furthermore, the second part concentrates on enhancing the robustness of the distributed inference system against the possible unavailability of edge devices. This involves developing fault-tolerant mechanisms that ensure the system remains operational and that inference processes produce accurate results even in the face of device failures or connectivity issues. The main work in this part presents RobustDiCE, a strategy that improves the robustness of distributed inference at the edge. RobustDiCE evaluates the importance of all neurons and evenly distributes critical neurons across multiple devices, ensuring that the model maintains comparable accuracy even in the face of potential device failures. By partially replicating only the most important neurons, instead of full replication, RobustDiCE ensures that these critically important neurons remain operational to the greatest possible extent in the event of device failures. Lastly, the thesis explores EASTER, a similar partitioning method designed for large language models to balance resource utilization and model robustness against device failures.

To conclude, this thesis presents efficient and robust solutions for deploying advanced DNN models at the edge. These methods optimize resource utilization by minimizing memory usage per device, reducing energy consumption, and potentially improving overall system throughput. They also enhance the reliability and robustness of deployments in the face of device failures or connectivity issues, ensuring that the system continues to operate reliably and retains the accuracy of distributed inference. Each method developed and presented in this thesis contributes to the more widespread adoption of intelligent, distributed edge AI in resource-constrained environments.

SAMENVATTING

Naarmate diep neurale netwerken (DNN's), vooral grote taalmodellen, steeds complexer en omvangrijker worden, nemen hun rekenbehoeften toe. DNN-gebaseerde applicaties worden doorgaans als diensten aangeboden op cloud-servers die zijn uitgerust met talrijke GPU's. Door de snelle uitbreiding van Internet of Things (IoT)-netwerken, gekenmerkt door hun talrijke verbonden apparaten, heeft deze ontwikkeling een paradigmaverschuiving noodzakelijk gemaakt naar het verwerken van gegevens dichter bij de bron in plaats van in gecentraliseerde clouds. Dit bevordert een grote interesse in het implementeren van diepe leermodellen aan de edge. De term "edge" verwijst hier naar een verscheidenheid aan genetwerkte apparaten met , geplaatst ergens langs het pad van gegevensoverdracht tussen een gegevensbron en de cloud.

Het implementeren van DNN's aan de edge biedt verbeterde privacy, veiligheid en betrouwbaarheid, waardoor deze aanpak aantrekkelijk maar uitdagend is voor gebruikerstoepassingen. De belangrijkste uitdaging komt voort uit de intensieve rekenbehoeften van neurale netwerken, terwijl edge-apparaten meestal beperkte beschikbare bronnen hebben. Dit verschil vereist innovatieve oplossingen zoals modelcompressie, destillatie, enz. om complexe DNN-modellen effectief binnen de beperkingen van edge-omgevingen uit te voeren. Deze methoden kunnen echter iteratieve retrainingskosten introduceren voor specifieke gebruikerstoepassingen en potentiële in bepaalde beeld of natuurlijke taalverwerkingstaken.

Deze scriptie richt zich op de gezamenlijke uitvoering van grote DNN's over meerdere edge-apparaten, waarbij strategieën worden onderzocht voor efficiënte en robuuste modelimplementatie op bronbeperkte edge-apparaten. De scriptie is verdeeld in twee hoofdonderdelen:

Het eerste deel richt zich op het oplossen van de uitdagingen die worden opgeworpen door de beperkte rekenbronnen van edge-apparaten. Het omvat het ontwikkelen van systemen en hulpmiddelen die de snelle en efficiënte gedistribueerde implementatie van DNN-modellen over meerdere edge-apparaten vergemakkelijken. We ontwerpen het AutoDiCE-framework voor het automatiseren van het distributieproces. Dit framework vereen-

voudigt de partitionering van modellen, automatiseert codegeneratie voor gedistribueerde uitvoering en optimaliseert inter-apparaatcommunicatie. Daarnaast verkennen we ook optimale distributieconfiguraties van DNN-modellen voor verschillende doelstellingen. Een geavanceerde techniek voor het verkennen van ontwerpruimte (DSE), die gebruik maakt van nieuwe genetische codering, zoekt efficiënt naar optimale distributiestrategieën van om energie- en geheugengebruik te minimaliseren en tegelijkertijd de systeemdoorvoer te maximaliseren.

Verder concentreert het tweede deel zich op het versterken van de robuustheid van het gedistribueerde inferentiesysteem tegen de mogelijke niet-beschikbaarheid van edge-apparaten. Dit omvat het ontwikkelen van fouttolerante mechanismen die ervoor zorgen dat het systeem operationeel blijft en dat de inferentieprocessen nauwkeurige resultaten blijven produceren, zelfs in het geval van apparaatstoringen of connectiviteitsproblemen. Het belangrijkste werk in dit deel presenteert RobustDiCE, een strategie die de robuustheid van gedistribueerde inferentie aan de edge verbetert. RobustDiCE evalueert het belang van alle neuronen en verdeelt kritieke neuronen gelijkmatig over meerdere apparaten, zodat het model vergelijkbare nauwkeurigheid behoudt, zelfs in het geval van mogelijke apparaatstoringen. Door alleen de belangrijkste neuronen gedeeltelijk te repliceren, in plaats van volledige replicatie, zorgt RobustDiCE ervoor dat deze kritisch belangrijke neuronen in de grootst mogelijke mate operationeel blijven bij apparaatstoringen. Ten slotte verkent de scriptie EASTER, een vergelijkbare partitioneringsmethode ontworpen voor grote taalmodellen om resourcegebruik en modelrobustheid tegen apparaatstoringen in balans te brengen.

Samengevat presenteert deze scriptie efficiënte en robuuste oplossingen voor het implementeren van geavanceerde DNN-modellen aan de edge. Deze methoden optimaliseren het gebruik van bronnen door het geheugengebruik per apparaat te minimaliseren, het energieverbruik te verlagen en mogelijk de algehele systeemdoorvoer te verbeteren. Ze verbeteren ook de betrouwbaarheid en robuustheid van implementaties in het geval van apparaatstoringen of connectiviteitsproblemen, waardoor het systeem betrouwbaar blijft werken en de nauwkeurigheid van gedistribueerde inferentie behoudt. Elk ontwikkelde en gepresenteerde methode in deze scriptie draagt bij aan de bredere adoptie van intelligente, gedistribueerde edge AI in bronbeperkte omgevingen.

"To live intensely and richly, merely to exist, that depends on ourselves."

— Born to Win

ACKNOWLEDGMENTS

The boldest decision I made in recent years was to live abroad independently and pursue a PhD in the Netherlands. I had not anticipated the COVID pandemic, which engulfed Europe just under two months after I started my PhD in 2020. Pursuing a PhD was not merely a commitment of time; it was an adventure. This process is challenging for many, but 'effort' conquers all—not erratic or misdirected effort, but consistent, dedicated effort. These efforts helped me to overcome loneliness, frustration, and the discouragement of unsuccessful submissions. Even now, I am amazed that I have completed my PhD journey.

First of all, my most sincere gratitude goes to my supervisors, Professor A.D. Pimentel and Dr T.P. Stefanov, for taking me on this long journey. Andy, a distinguished academic in computer systems, is renowned for his wisdom and passion. He granted me enough freedom to explore and the patience necessary for rigorous research, profoundly inspiring my work. His communication skills and mastery of project management have provided meaningful benefits to me. Todor is an expert in embedded systems, renowned for his lifelong dedication to learning new knowledge. His comprehensive guidance on writing and his foresight in anticipating potential challenges in my research have been invaluable, particularly in shaping the narrative of my research. Coming from a background devoid of academic writing experience, I had to learn the ropes and revise my articles from scratch. Andy and Todor have demonstrated remarkable patience with me throughout the writing of academic articles and my Ph.D. thesis. They meticulously revised my work sentence by sentence, paragraph by paragraph, and section by section, engaging in fruitful discussions along the way. They not only taught me the essentials of scientific and accurate writing, but also how to maintain consistency and fluency in my scientific expressions, ensuring logical coherence. Their efficiency in revising drafts is exceptionally high, often aiding me in meeting submission deadlines during late nights or weekends. I am deeply moved and grateful for their dedication; without their genuine support and infectious enthusiasm for academia, defending my thesis on time would have been mission impossible.

It is my lifelong honor that Prof. R.V. van Nieuwpoort, Prof. D. Müller-Gritschneder, Prof. ir. A. Iosup, Prof. P. Grosso, Dr. A. Pathania, and Dr. D. Sapra accepted the invitation to become my Ph.D.committee member. I would like to extend my gratitude to them for reading this thesis and providing their invaluable feedback.

Thanks to Juriaan (LIACS), Simon, and Martin for their assistance in both daily work and personal matters, such as managing computing clusters, purchasing working items, guiding me in Dutch, etc. Chatting with them in the office is always a relaxing and enjoyable experience. Additionally, I would like to thank our secretaries— Nusa (LIACS), Grace, Petra, and Nicole — for their efficiency and helpfulness in managing academic affairs, extending contracts, and processing visa applications. Additionally, I would like to thank my colleagues in the PCS research group, for all the social events, coffees, presentations, and discussions.

Especially to Ana O, Ana V, Andrés, Anuj, Benny, Benny, Clemens, Daphne, Dolly, Ehsan, Francesco, Jelle, Julius, Jun, Lukas, Marco, Marius, Pooya, Saeedeh, Shaoshuai, Sudaksh, Sudam, Uraz, and Yixian. I was also lucky to work in the LIACS. A big thanks to Svetlana and Erqian for their help, and also for the shared time, discussions, and project-related programming experiences. I will always enjoy having fun in many casual conversations with Abolfazi, Faezeh, Fatemeh, Peng Wang, Roozbeh, Sobhan, and Sumiran.

Special thanks to my brilliant friend, Qi Wang. I am fortunate to know Qi and to learn from him. Without Qi, my research might have remained mired in engineering, neglecting the importance of narrative storytelling. Qi inspired me to develop research topics focused on scenario-based problems and solutions driven by algorithms, rather than relying heavily on extensive engineering efforts.

Particularly, I would like to extend my heartfelt thanks to Kexin Wang, QianQian Jia, Jiangyun Hou, Shang Shi, Kena Zheng, Weijian Liu, and Zhong Li for companying and for helping me during my difficult time. Thanks to Danru Xu, Gaosheng Liu, Kexin Liang, Ming Li, Shiqi Liu, Tao Hu, Wei Wang, Weijie Wei, Xiaofei Yu, Xiaoyu Tong, Xiayu Zhang, Yahui Zhang, Yongtuo Liu, Yue Chen, Zenglin Shi. I cherish the time spent with them playing board games, enjoying Chinese food, and sharing leisure moments. Thanks to Biwen Wang, Chao Xu, Cong Liu, Di Wu, Hui Chang, Hui Feng, Jian Dong, Ji He, Jiayi Shen, Jiayang Shi, Jie Liu, Jingwen Jia-Zhang, Jingwen Liao, Jun Xiao, Lin Zhang, Li Xu, Lu Zhang, lvqi Liu, Menglin Wu, Na Li, Pengwan Yang, Puzhong Zhang, Qi Bi, Qianru Zuo, Qinyu Chen, Renjie Lv, Ruihong Yin, Ruyue Xin, Shenghao Qiu, Shirley, Shuang Su, Shuangyi Zhao, Siyu Li, Tiantianlang Wang, Tonghui Yin, Weijia Zhang, Weikang Weng, Wenyang Wu, Wenzhe Yin, Xiaojian Du, Xiaoyan Xing, Xinlu Chen, Xinwei Liu, Xinyu Zhang, Yachen Liu, Yaozu Han, Ye Liu, Yingjun Du, Yuandou Wang, Yuxuan Zhao, Yue Li, Yue Song, Yunlu Chen, Zehao Xiao, Zhangyu Xiao, Zeshun Shi, Zhao Yang, Ziyuan Wang, Zong Fan and many other friends for engaging in academic discussions, sharing life stories, playing basketball, and enjoying funny moments together.

Finally, I must express profound gratitude to my parents, brother, and sister-in-law for their support. Throughout my childhood and many challenges in life, you have always allowed me to be myself. The COVID pandemic prevented us from celebrating traditional Chinese festivals together, but distance cannot keep our hearts apart. I am deeply grateful to know my wife, Zhang Jia. I treasure every moment with you and I look forward to spending the rest of my life with you.

There have been many people who have supported me over the past few years, and despite my best efforts to list everyone's names, I may have missed some! I apologize if your name should have been mentioned; I remain grateful for your help and support. This essay is also dedicated to all of you.