Efficient Machine Learning for Edge Computing: Architecture and Application

WENYAO ZHU

Doctoral Thesis
Stockholm, Sweden, 2025

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Electronics and Embedded Systems
SE-16640 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik fredagen den 28 mars 2025 klockan 14.00 i Sal C, Kistagången 16, Kungliga Tekniska Högskolan, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Machine learning has demonstrated exceptional capability in solving complex tasks across a wide range of fields. Advances in hardware accelerators have enabled the deployment of machine learning models on edge devices, facilitating real-time AI applications in resource-constrained systems. Recent accelerators have increasingly adopted Network-on-Chip (NoC) architectures to support massive data communication within large-scale processing element arrays. However, as the complexity of these accelerators continues to grow, effective design-space exploration before hardware prototyping becomes essential. Additionally, achieving high flexibility and efficiency across diverse machine learning workloads remains a significant challenge, especially for edge computing.

To address these problems, we explore from both the architecture side and the application side. Firstly, we introduce a cycle-accurate simulation tool for NoC-based deep neural network (DNN) accelerators. This simulator enables rapid and precise evaluation of inference efficiency by exploring design parameters. By providing detailed performance tracing into system behavior, the simulator facilitates the optimization of DNN inference efficiency, which can reduce the time and cost associated with hardware prototyping. Then we focus on novel architectural designs for NoC-based DNN accelerators, leveraging in-network processing techniques to improve end-to-end latency and resource utilization. Two key approaches are proposed: an activation-in-network design that offloads non-linear operations to the NoC and a pooling on-the-go design that minimizes communication overhead for pooling layers. These designs demonstrate substantial improvements in processing efficiency upon existing NoC based accelerator architectures, while maintaining scalability and adaptability for diverse DNN workloads.

The third part explores the application of machine learning in embedded sensor systems, with a focus on lower-limb prostheses. A wearable pressure measurement system is developed to collect and analyze intra-socket pressure data. Two machine learning applications are proposed for solving sub-tasks within the field of comfortable prosthetic socket design. A clustering-based method is developed for optimizing sensor deployment by reducing redundancy while maintaining data integrity. A gait phase recognition approach that utilizes multiple hidden Markov models and Gaussian mixture models is developed. The proposed gait recognition method achieves high accuracy and computational efficiency, which outperforms conventional techniques.

By tackling the challenges in NoC-based accelerator design and machine learning applications for embedded systems, we bridge the gap between hardware optimization and practical deployment. These techniques would pave the way for future advancements in edge intelligence.

Keywords: Network-on-Chip, Neural Network Accelerator, Accelerator Performance Simulation, In-Network Processing, Embedded Sensor System, Machine Learning for Prosthetics

Sammanfattning

Maskininlärning har visat en enastående förmåga att lösa komplexa uppgifter inom en mängd olika områden. Framsteg inom hårdvaruacceleratorer har möjliggjort implementeringen av maskininlärningsmodeller på kant-enheter, vilket underlättar realtidsapplikationer för AI i resursbegränsade system. Nya acceleratorer har i ökande grad använt Network-on-Chip (NoC)-arkitekturer för att stödja massiv datakommunikation inom storskaliga bearbetningsenhetsarrayer. Men i takt med att dessa acceleratorers komplexitet fortsätter att öka, blir effektiv designutforskning innan hårdvaruprototyputveckling avgörande. Dessutom kvarstår utmaningen att uppnå hög flexibilitet och effektivitet för olika maskininlärningsarbetsbelastningar, särskilt för kantberäkningar.

För att hantera dessa problem undersöker vi lösningar både från arkitektursidan och applikationssidan. För det första introducerar vi ett cykelnoggrant simuleringsverktyg för NoC-baserade djupa neurala nätverksacceleratorer (DNN). Detta simuleringsverktyg möjliggör snabb och noggrann utvärdering av inferenseffektivitet genom att utforska designparametrar. Genom att tillhandahålla detaljerad prestandaspårning av systembeteende underlättar simulatorn optimeringen av DNN-inferenseffektivitet, vilket kan minska tiden och kostnaden som är förknippad med hårdvaruprototyputveckling. Därefter fokuserar vi på nya arkitekturdesigns för NoC-baserade DNN-acceleratorer och använder tekniker för bearbetning inom nätverket för att förbättra slut fördröjningen och resursutnyttjandet. Två nyckelmetoder föreslås: en äctivation-in-networkdesign som flyttar icke-linjära operationer till NoC och en pooling on-the-godesign som minimerar kommunikationsöverliggning för poolinglager. Dessa designlösningar uppvisar betydande förbättringar i bearbetningseffektivitet jämfört med befintliga NoC-baserade acceleratorarkitekturer samtidigt som de behåller skalbarhet och anpassningsförmåga för olika DNN-arbetsbelastningar.

Den tredje delen utforskar maskininlärningens tillämpningar i inbyggda sensorsystem, med särskilt fokus på proteser för nedre extremiteter. Ett bärbart tryckmätningssystem utvecklas för att samla in och analysera tryckdata inom protesens sockel. Två maskininlärningsapplikationer föreslås för att lösa deluppgifter inom området för bekväm design av protes-socklar. En klusterbaserad metod utvecklas för att optimera sensorplacering genom att minska redundans samtidigt som dataintegriteten bibehålls. En gångfasigenkännings metod som använder flera dolda Markovmodeller och Gaussiska blandningsmodeller utvecklas. Den föreslagna gångigenkänningsmetoden uppnår hög noggrannhet och beräkningsmässig effektivitet, vilket överträffar konventionella tekniker.

Genom att hantera utmaningarna inom NoC-baserad acceleratorarkitektur och maskininlärningsapplikationer för inbyggda system överbryggar vi gapet mellan hårdvaruoptimering och praktisk implementering. Dessa tekniker banar vägen för framtida framsteg inom kantintelligens.

Nyckelord: Network-on-Chip, Neurala Nätverksacceleratorer, Prestandasimulering för Acceleratorer, Bearbetning inom Nätverket, Inbyggt Sensorsystem, Maskininlärning för Proteser

List of Papers

I **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "NoCDAS: A Cycle-Accurate NoC-based Deep Neural Network Accelerator Simulator", manuscript submitted to *ACM Transactions on Modeling and Computer Simulation*, (2024).

II **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "Activation in Network for NoC-Based Deep Neural Network Accelerator", in *Proceedings of International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA)*, HsinChu, Taiwan, (2024).

III **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "Pooling On-the-go for NoC-based Convolutional Neural Network Accelerator", in *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, (2024).

IV Zhonghai Lu, **Wenyao Zhu**, Yizhi Chen, Josephine Charnley, Valter Dejke, Andrii Pomazanskyi, Siu-Teing Ko, Begum Zeybek, Pouyan Mehryar, Zulfiqur Ali, Michalis Karamousadakis, and Dejiu Chen, "Wearable Pressure Sensing for Lower Limb Amputees", in *Proceedings of IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Taipei, Taiwan, (2022).

V **Wenyao Zhu** and Zhonghai Lu, "Evaluation of Time Series Clustering on Embedded Sensor Platform", in *Proceedings of 24th Euromicro Conference on Digital System Design (DSD)*, Palermo, Italy, (2021).

VI **Wenyao Zhu**, Yizhi Chen, Siu-Teing Ko, and Zhonghai Lu, "Redundancy Reduction for Sensor Deployment in Prosthetic Socket: A Case Study", *Sensors*, 22, no. 9: 3103, (2022).

VII **Wenyao Zhu**, Zhenbang Liu, Yizhi Chen, Dejiu Chen, and Zhonghai Lu, "Amputee Gait Phase Recognition Using Multiple GMM-HMM", *IEEE Access*, vol. 12, (2024).

List of acronyms

AI	Artificial Intelligence
AiN	Activation in Network
ASIC	Application Specific Integrated Circuit
BMU	Best Matching Unit
CMP	Chip multiprocessor
DNN	Deep Neural Network
DTW	Dynamic Time Warping
EMG	Electromyography Sensors
FPGA	Field Programmable Gate Array
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HLS	High Level Synthesis
HMM	Hidden Markov Model
IMU	Inertial Measurement Unit
JSD	Jenson–Shannon Divergence
LSTM	Long Short-term Memory
MAC	Multiply and Accumulate
MC	Memory Controller
NI	Network Interface
NoC	Network-on-Chip
NoC-DNN	Network-on-Chip Based DNN Accelerator
NoP	Network-on-Package
RTL	Register Transfer Level
PCC	Pearson correlation coefficient
PE	Processing Element
SoC	System-on-Chip
SOM	Self-Organizing Map
TOPS	Tera Operations Per Second
VC	Virtual Channel

Contents

Chapter 1

Introduction

1.1 Background

Machine learning addresses the problem of enabling computers to automatically learn from data and make decisions based on experience and observations [1]. Over recent years, machine learning methods have emerged as the foundation of powerful tools for tackling more complex tasks. With advances in innovative architectures and algorithms, the rapid increase in the availability of training data, and the development of semiconductor technology, deep learning has popped out among numerous machine learning methods. The evolution of deep neural networks (DNNs) has significantly advanced the state-of-the-art in many real-world artificial intelligence (AI) applications such as visual pattern recognition, object detection, and natural language processing [2–4].

However, the quest for heightened accuracy and performance in DNN models necessitates deeper and more complex architectures, leading to increased computation costs. The first DNN model to leverage GPU acceleration for training, AlexNet [5], introduced a decade ago, comprised approximately 60 million parameters. Nowadays, the mainstream large language model GPT-4 [6] contains hundreds of billions of parameters.

The ongoing explosion of DNN model sizes has made them computationally intensive and costly. Achieving high efficiency and reduced latency has become critical in DNN applications. Moreover, the rising demand for AI on embedded systems such as mobile phones, autonomous vehicles, and smart healthcare devices, poses a significant challenge to their deployment in daily life.

1.1.1 DNN hardware accelerators

DNNs typically comprise millions of neurons, each executing a vast number of arithmetic operations, such as multiply-and-accumulation (MAC) operations and non-linear operations. The nature of simple but large-scale computations involved in DNNs has gone beyond the real-time processing capability of current resource-

constraint embedded devices based on general-purpose CPUs [7]. As a result, hardware accelerators that enable large-scale parallel processing become the practical solution for efficient DNN inference [8].

Many DNN hardware accelerators have been developed to accelerate machine learning applications on edge. Based on different hardware platforms, these accelerators mainly lie in three categories: GPU (aim for general-purpose applications), FPGA (aim for a group of models), and ASIC (aim for a specific task). With the increased design cost and reduced flexibility, power efficiency and latency are improved from GPU to ASIC-based accelerators.

GPUs provide high runtime reconfigurability, allowing them to support a wide range of DNN applications. With the help of machine learning frameworks and GPU programming libraries such as PyTorch [9] and CUDA [10], the quick adjustment and verification of DNN architectures can be realized. While offering high throughput with large-scale parallel processing units, GPU generally consumes much more power than the other two platforms. It performs well in processing DNN tasks with large batches of data but can have higher latency for small-scale inferences. That is due to the kernel setup and data transfer overhead, and the lower PE utilization. Thus GPU may not be a suitable hardware for efficient machine learning on edge.

FPGAs are attractive platforms for DNN acceleration on edge devices due to their low latency and high energy efficiency compared to GPUs. Various DNN accelerator implementations have been implemented on embedded FPGA for image classification and pattern recognition [11–13], showing improved performance and reduced power consumption. Though FPGA can reconfigure the programmable gates to support flexibility accelerator architecture design, the limitation of computational resources in edge-scale FPGA devices makes DNN deployment challenging. DNNWeaver [14] is developed to generate FPGA accelerator instances commensurate with the high-level DNN model. It uses hand-optimized design templates with a recurrent structure to maximize data reuse. To balance the model complexity and available FPGA resources, software and hardware co-design methodology has been proposed [15]. The design flow includes the hardware-aware neural network architecture search [16], a tile-based accelerator template, and an HLS tool for generating synthesizable code. To further mitigate the implementation cost of FPGA accelerators, Zhang *et al.* [17] developed a tool that can automatically build high-performance DNN accelerators on FPGAs to meet the latency and throughput requirements. These tools enable fast DNN application deployment on edge.

Customized ASIC chips have been developed to further improve performance and energy efficiency in specific DNN tasks. Recent ASIC-based DNN accelerators such as ShiDianNao [18], Eyeriss [19], Cambricon-X [20], and TPU [21] adopt large-scale processing element (PE) arrays to enable concurrent execution. They use optimization techniques such as row stationary dataflow, sparse skip data transfer, and bit-width adaptive computing for better latency and power consumption in massive tensor processing. However, due to rigidly designed data paths, their applicability is often limited to particular types and shapes of DNN layers.

Fig. 1.1 summarizes the aforementioned DNN accelerator platforms. GPUs

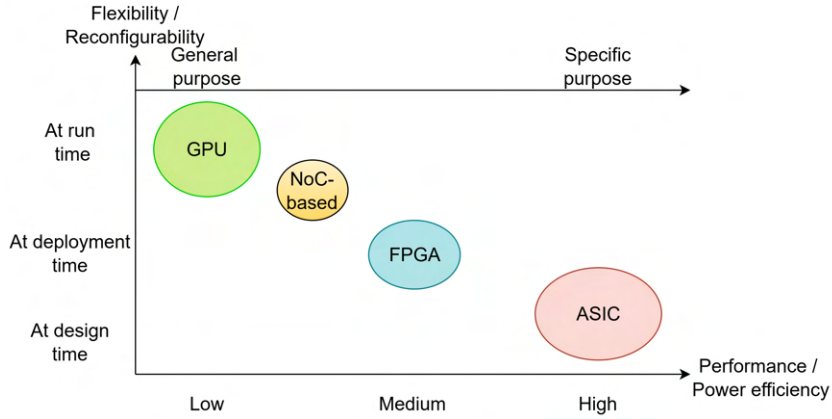


Figure 1.1: DNN hardware accelerator overview. (Adapted from [22])

offer a high reconfigurability feature at runtime, supporting diverse DNN applications. However, these platforms suffer from high power consumption and data transfer latency between processing units and memory. In contrast, the ASIC-based DNN accelerators are specifically designed for a particular DNN model in order to achieve optimal performance and power efficiency. In return, these designs limit the computational flexibility. As a result, the ASIC-based DNN accelerators are not reconfigurable at design time.

Compared with ASIC-based accelerators, FPGA provides a flexible platform for deploying various DNN accelerator designs because of its reconfigurable feature. As shown in **Fig.** 1.1, although the FPGA-based designs improve the design flexibility, the computational flexibility is insufficient to support reconfigurability at runtime. For example, the data path is fixed for a particular RNN or DNN model at design time, and no further reconfiguration can be done at runtime.

As the real DNN tasks can combine multiple models and may adjust during processing, the demand for computational flexibility has increased [22]. Therefore, runtime reconfigurability becomes one of the key factors in novel accelerator design. In this case, interconnection turns into an essential design consideration that supports data transmissions in the accelerator.

1.1.2 Network-on-Chip based DNN accelerator

As Moore's Law slows and Dennard Scaling reaches its end due to physical limits and thermal constraints in transistor technology, scaling the performance of single-core processors has become increasingly difficult [23]. Chip multiprocessors (CMPs) have been the prevalent microprocessor design approach for nearly two decades. With advancements in technology nodes, CMPs can now incorporate hundreds of cores on a single chip. The on-chip interconnection concept, Network-

on-Chip (NoC) [24], has been proposed to meet the concurrent requirement of data transmission in CMP [25].

Modern DNN hardware accelerators typically consist of multiple PEs, each configured with the local memory, execution unit, control unit, etc. These PEs can be treated as simplified microprocessor cores, and the accelerator is organized similarly to CMP. Therefore, NoC is also a suitable interconnection scheme for DNN hardware accelerators. Due to its better scalability than the bus, the NoC-based architecture can enlarge the PE array size to achieve higher computing performance on a single chip. NoC is a potential solution for supporting efficient and reliable data transfer in DNN hardware accelerators.

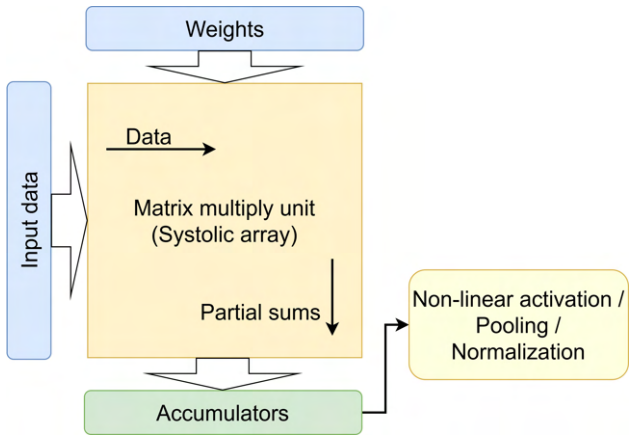


Figure 1.2: Systolic array in TPU [21].

The systolic array was originally proposed as a highly parallel structure for computationally intensive applications [26]. It has recently been applied to accelerate DNN. The first Tensor Processing Unit (TPU) from Google [21] uses the systolic array in its matrix multiply unit for handling common matrix operations in DNN workloads, which involves 256×256 MAC elements in total. As shown in **Fig.** 1.2, the weights are loaded from the top and the input data enters from the left. The operation moves diagonally across the matrix, where each weight sequentially interacts with incoming data. In this way, the energy cost of the memory is reduced with fewer reads and writes. On the other side, the rigidly defined data flow requires each of the elements in the systolic array to proceed in lock steps. So it is good for repeated workloads with computation bound, such as convolution layers. However, the utilization is lower for other DNN tasks that need higher memory bandwidth, such as fully connected and recurrent layers.

Recent NoC architectures address the limitations in such rigidly designed homogeneous systolic arrays. By enabling flexible communication within the PE array, they mitigate the gaps in hardware utilization due to the different functions and layer shapes in various workloads. NoC connects numerous PEs and other core

types using an arrangement of links and routers, facilitating dynamic data transmission in the form of packets. This decouples data transfer from the computation process within the PEs, allowing for more efficient utilization of hardware resources.

NoC-DNN (Network-on-Chip based DNN accelerator) can efficiently support flexible traffic patterns for various DNN data flows, thereby enhancing computational flexibility. Thus it serves a critical role in the overall performance of the DNN execution [7]. Several NoC-DNN architectures have been designed [27–31] to support a wide range of DNNs on a single chip. The following subsections introduce three prominent NoC-DNN architectures.

1.1.2.1 MAERI

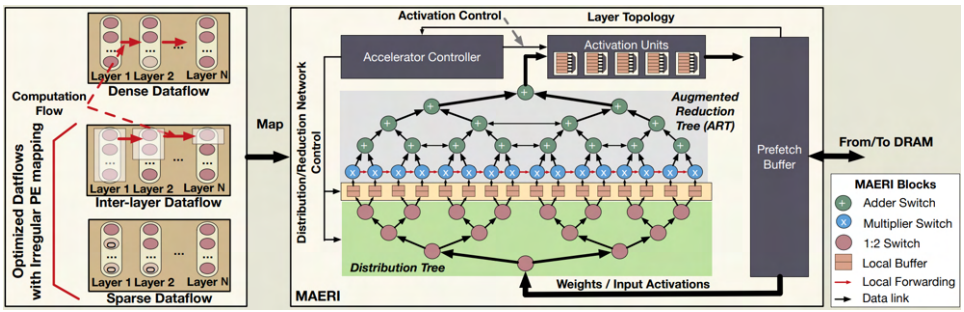


Figure 1.3: MAERI architecture (Georgia Tech). [28]

The MAERI (Multiply-Accumulate Engine with Reconfigurable Interconnects) architecture uses a modular approach instead of fixed interconnect designs. As shown in **Fig.** 1.3, there are two configurable network modules: a distribution network for broadcasting input activations and weights, and a reduction network for efficiently accumulating partial sums. These networks, built with lightweight configurable switches, enable dynamic adaptation to a wide range of layer types and dataflows, including convolutional, recurrent, pooling, and fully connected layers.

This flexible NoC design allows MAERI to achieve high resource utilization and adapt to both dense and sparse computations. It demonstrates significant improvements in throughput, with up to 4.5× better utilization across multiple dataflow mappings over baselines accelerators with rigid interconnection fabrics, while still maintaining area and power efficiency. MAERI enables the efficient construction of arbitrarily sized MAC engines, creating opportunities to support novel optimizations in DNNs.

1.1.2.2 Eyeriss-v2

The Eyeriss-v2 architecture introduces a flexible and efficient NoC design, known as the hierarchical mesh network, to address the challenges of executing compact

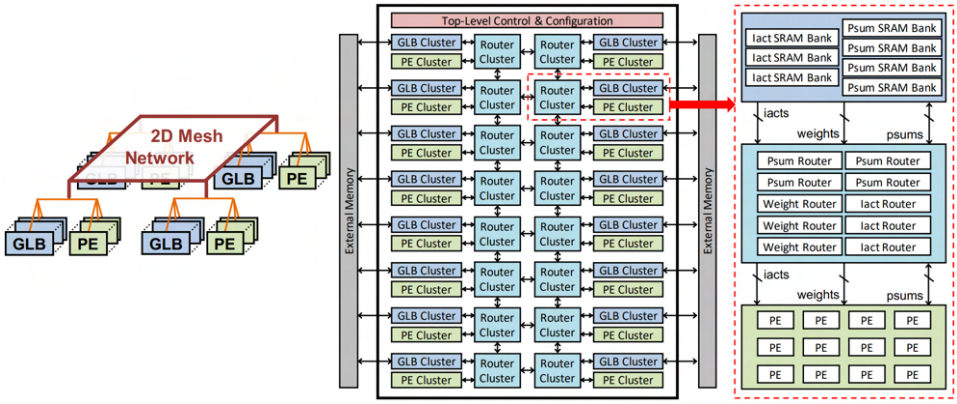


Figure 1.4: Eyeriss-v2 architecture (MIT). [29]

and sparse DNNs. As shown in **Fig.** 1.4, unlike traditional flat multicast networks, the hierarchical NoC employs a two-level hierarchy: clusters of PEs are connected through an all-to-all network at the local level, while clusters themselves are linked via a global 2D mesh network. This structure enables high bandwidth for data transfer when reuse is low and leverages spatial data reuse when available, ensuring efficient handling of diverse DNN workloads.

Eyeriss-v2 achieves significant improvements in both performance and energy efficiency. For sparse MobileNet, it delivers a throughput of 1470.6 inferences per second and an energy efficiency of 2560.3 inferences per joule, outperforming the original Eyeriss by 12.6× in speed and 2.5× in energy efficiency. This adaptability to varying bandwidth and reuse requirements makes it a standout architecture for emerging compact DNNs.

1.1.2.3 Simba

The Simba architecture employs a hierarchical interconnect combining a NoC for intra-chiplet communication and a Network-on-Package (NoP) for inter-chiplet communication, enabling scalable and efficient execution of DNNs. This design, as depicted by **Fig.** 1.5, leverages a tile-based approach where chiplets are interconnected in a mesh topology, facilitating flexible communication patterns for diverse DNN workloads. The integration of unicast and multicast capabilities within the NoC and NoP ensures effective data transmission, while hybrid wormhole and cut-through flow control minimize latency and avoid deadlocks.

Simba achieves up to 128 TOPS with energy efficiency ranging from 0.2 to 6.1 TOPS/W, depending on operational voltage. Through tailored tiling strategies and communication-aware optimizations, it minimizes the overheads associated with inter-chiplet communication, delivering high performance across a range of DNN tasks, including low-latency inference for models like ResNet-50.

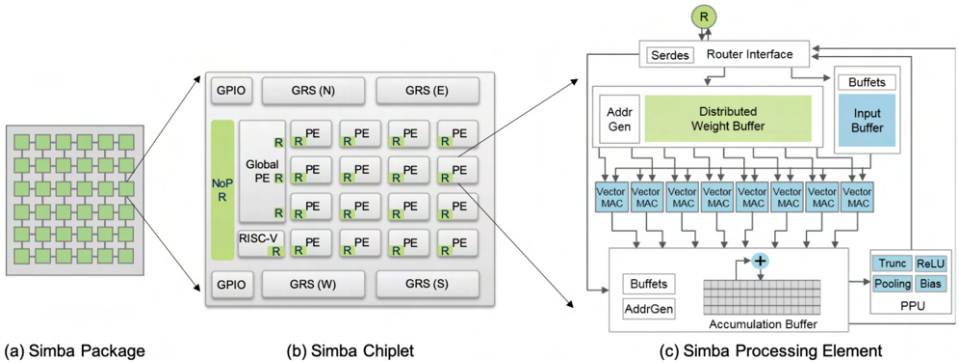


Figure 1.5: Simba architecture (Nvidia). [30]

1.2 Challenges for efficient machine learning on edge

1.2.1 Architectural challenges: NoC-based DNN accelerator designs

NoC-based DNN accelerators offer significant advantages in flexibility, enabling the efficient execution of diverse neural network workloads on a single platform. This adaptability arises from the capability of NoC designs to dynamically manage data flows, balance computational loads, and support heterogeneous data patterns. However, achieving this flexibility while maintaining scalability, performance, and energy efficiency introduces substantial challenges. Key issues include evaluating the impact of network scaling, balancing hardware utilization for various DNN layers, and optimizing traffic patterns and scheduling PE tasks.

Performance evaluation of NoC-DNNs is essential to understand and address these challenges. To efficiently explore the NoC-DNN design space and reduce hardware design time and cost, it is necessary to utilize a NoC-DNN simulator before building and testing the real hardware system. Simulations can also uncover bottlenecks in the system caused by high latency, uneven resource utilization, or insufficient bandwidth, especially for workloads with complex and varying computational demands.

One promising solution to these challenges is integrating novel NoC-DNN architectural designs that incorporate advanced processing capabilities. For example, by enabling in-network processing as a fundamental component, these designs can reduce data movement and communication overhead. The localized processing alleviates the computational burden on PEs to reduce latency. These advancements are critical for developing scalable and efficient machine learning accelerators that meet the stringent requirements of edge computing.

1.2.2 Application challenges: Machine learning for embedded sensor systems

Machine learning applications for embedded sensor systems face unique challenges, both in analyzing the data they generate and in deploying solutions tailored to their constraints. These systems often produce time-sensitive data that must be processed with high accuracy and robustness, whether locally on embedded platforms or remotely on more powerful systems.

In many cases, embedded sensor systems operate in environments with intermittent connectivity, making it necessary to ensure data reliability and analysis continuity locally. Additionally, the diverse and often noisy nature of sensor data requires sophisticated preprocessing and modeling approaches to extract actionable features effectively. While lightweight models and adaptive algorithms are essential for on-device processing, the analysis of sensor data often benefits from advanced machine learning techniques executed on external platforms, where computational resources and energy constraints are less critical. Addressing this challenge would leverage novel machine learning solutions for comprehensive data analysis. Together, these efforts enable accurate and efficient solutions for real-world edge applications.

1.3 Research contribution

This thesis addresses the challenges of designing and optimizing NoC-based DNN accelerators and applying machine learning to embedded sensor systems. The research advances both the architectural innovations needed for efficient edge computing and the methodologies for analyzing data for embedded sensor platforms. Specifically, the contributions of this thesis are as follows:

1. Simulation tool for NoC-DNNs: A detailed open source simulation framework is developed to explore the NoC-DNN design space. The simulator identifies bottlenecks in latency and resource utilization, providing insights that guide the optimization of traffic patterns, hardware scaling and PE task mapping. The research contribution is presented in the following manuscript:

    - **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "NoCDAS: A Cycle-Accurate NoC-based Deep Neural Network Accelerator Simulator", submitted to *ACM Transactions on Modeling and Computer Simulation*, (2024).

2. Novel NoC-DNN architectural designs: Based on the simulation of the common NoC-DNN, this thesis proposes and evaluates innovative NoC architectures that integrate in-network processing to reduce data movement and alleviate computational bottlenecks. The first design leverages the idea of computation while blocking to offload non-linear activations to the network. The second design aims to do layer fusion for pooling operations to minimize

their communication overheads. The research contributions are published in the following papers:

- **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "Activation in Network for NoC-Based Deep Neural Network Accelerator", in *Proceedings of International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA)*, HsinChu, Taiwan, (2024).
- **Wenyao Zhu**, Yizhi Chen, and Zhonghai Lu, "Pooling On-the-go for NoC-based Convolutional Neural Network Accelerator", in *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Samos, Greece, (2024).

3. Machine learning applications for embedded sensor systems: This part contributes to the development of advanced methodologies for wearable pressure measurement systems and machine learning applications aimed at enhancing prosthetic socket design and functionality for lower-limb amputees. The embedded sensor system design and a feasibility study of clustering algorithms running on the system is first discussed, and the research contributions are published in the following papers:

   - Zhonghai Lu, **Wenyao Zhu**, Yizhi Chen, Josephine Charnley, Valter Dejke, Andrii Pomazanskyi, Siu-Teing Ko, Begum Zeybek, Pouyan Mehryar, Zulfiqur Ali, Michalis Karamousadakis, and Dejiu Chen, "Wearable Pressure Sensing for Lower Limb Amputees", in *Proceedings of IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Taipei, Taiwan, (2022).
   - **Wenyao Zhu** and Zhonghai Lu, "Evaluation of Time Series Clustering on Embedded Sensor Platform", in *Proceedings of 24th Euromicro Conference on Digital System Design (DSD)*, Palermo, Italy, (2021).

   Two machine learning applications are then developed for data analysis in our embedded sensor system. A clustering-based analysis method is proposed to optimize sensor placement within prosthetic sockets by identifying and reducing redundancy in local areas. A hidden Markov model-based approach is developed for recognizing gait phases in lower-limb amputees using interfacial pressure data collected by wearable sensors. These methods collectively advance the field of machine learning applications in wearable sensor system for prosthetics. The research contributions are published in the following papers:

   - **Wenyao Zhu**, Yizhi Chen, Siu-Teing Ko, and Zhonghai Lu, "Redundancy Reduction for Sensor Deployment in Prosthetic Socket: A Case Study", *Sensors*, 22, no. 9: 3103, (2022).
   - **Wenyao Zhu**, Zhenbang Liu, Yizhi Chen, Dejiu Chen, and Zhonghai Lu, "Amputee Gait Phase Recognition Using Multiple GMM-HMM", *IEEE Access*, vol. 12, (2024).

1.4 Thesis organization

There are five chapters in this thesis. Chapter 1 gives an overall introduction of machine learning on edge. The rest of the thesis is organized as follows: Chapter 2 presents the simulation tool for NoC-based DNN accelerators and a comprehensive exploration of design parameters. Chapter 3 focuses on two efficient NoC-DNN architecture optimizations that leverage the in-network processing techniques. Chapter 4 investigates two machine learning applications for our embedded sensor system in the context of lower-limb prosthetics. Chapter 5 draws the conclusion and discusses the future work.

Chapter 2

NoC-based DNN accelerator simulator

2.1 Introduction

DNN accelerators involve sophisticated data flow and architecture design to ensure reconfigurability and efficiency, but the impact of the design parameter choices hasn't been well studied yet. Simulation tools can help architects explore the design space of NoC-DNNs, thereby reducing the design effort and cost. For example, the developers of Venus customize the analytical tool Timeloop [32] for performance comparison over other architectures.

2.1.1 Conventional DNN accelerator simulators

Conventional DNN accelerator simulators usually rely on abstract hardware architecture models to provide a coarse understanding of their performance characteristics when executing a wide range of DNNs.

SMAUG [33] is a DNN framework that supports end-to-end simulation for various DNN workloads on diverse accelerator modelings. However, its hardware architecture is limited to the backend simulation engine gem5-Aladdin [34], such as a systolic array or a convolution engine. So NoC-based accelerator structures can not be explored easily from SMAUG. On the other hand, since it is a trace-based simulator, it has to use a sampling approach for large DNNs to reduce the simulation time, which impedes the full model evaluation For NoC-DNN.

MAESTRO [35] is an analytical cost model that studies the data-centric mapping of DNN tasks to hardware resources. It can simulate diverse dataflow choices for a DNN model and hardware configuration and estimate the execution time and energy efficiency. The communication delay via NoC is estimated based on a pipe model involving bandwidth and average delay. Though their reported runtime is

This chapter is based on the material from paper 1.

close to the RTL simulation of two DNN accelerators, the detailed NoC configuration exploration is not supported.

2.1.2 NoC-DNN simulators

Unlike conventional DNN accelerator simulators, NoC-DNN simulators aim to implement a more detailed and flexible NoC hardware framework. They emulate both data transmission and neuron computation behaviors to provide insights into various design choices [36].

STONNE [37] serves as a backend in high-level DNN frameworks to simulate the inference process on a flexible DNN accelerator with modeled microarchitecture. It organizes three configurable components, the distribution network, the multiplier network, and the reduction network, to model the most current DNN accelerators. Though it supports various communication flows, its topology is limited to this three-tier architecture and lacks the support for canonical NoC interconnected PE structure.

Chen *et al.* [38] propose CNN-Noxim, a high-level cycle-accurate simulator for convolutional neural networks, and then improve it to DNNoC-sim [36] that introduces the cluster and slicing techniques for DNN models in addition to grouping. They have similar simulation workflow definitions, while DNNoC-sim supports small-sized NoC configurations as it doesn't need to map the DNN task at once.

These simulators excel in rapidly exploring optimization opportunities for DNN-NoC design but are short on comprehensively examining the architectural impact of DNN-NoC.

2.1.3 Level of detail in NoC-DNN simulators

A NoC simulator can be categorized by its levels of detail, which range from interface, capacity, flit, and hardware level [24].The interface level has the least network details to investigate the general behavior of NoC. In contrast, the flit level simulates very detailed microarchitectures of the network to trace individual data packets. Capacity level stands in between, which serves for initial performance assessment and quick reconfiguration. Normally the NoC simulator won't go to the hardware level to avoid unnecessary costs of time to build a physical implementation and run a simulation.

Similarly, these simulation levels also work for the NoC-DNN simulator designs. Simulation-based analytical tools such as NNest [39] and Timeloop [32] aim for early-stage design space exploration of DNN accelerators via behavioral simulation, which enables architectural trade-off evaluation across a wide range of applications. Thus they stay on the interface level and approximately quantify the accelerator performance and energy efficiency under different hardware constraints and mapping policies. They cannot perform NoC architecture exploration for NoC-DNN due to their limited range of architectural support.

The aforementioned NoC-DNN simulators, STONNE, CNN-Noxim, and DNNoC-sim, offer more detailed microarchitecture modules compared to analytical simulation tools. These simulators ensure the correct timing of DNN execution. On the other hand, they make abridged assumptions for NoC hardware. STONNE models the NoC-DNN components as three types of network building blocks. It simulates the behavior of each block individually and accumulates their executed cycles, but no detailed network traffic is considered. Therefore, STONNE is on the capacity level.

CNN-Noxim and DNNoC-sim employ the flit level NoC simulator Noxim [40] for accurate network behavior simulation. However, they assume that the PE can store and process a large number of neurons with a single data transfer, which simplifies the traffic in NoC for running DNN. Consequently, they stand between capacity and flit level.

These simulators cannot fully assess the exact performance of different NoC-DNNs. However, they have inspired us regarding the features and workflow of simulation. Based on that, we present our cycle-accurate NoC-based DNN accelerator simulator, called NoCDAS. It is a more realistic and flexible simulation platform at the flit level.

2.2 NoC-DNN simulator design

2.2.1 NoC-DNN hardware architecture

A NoC-based DNN hardware accelerator mainly involves three components: processing cores, memory, and the on-chip network. **Fig.** 2.1 illustrates the NoC-DNN accelerator architecture, which is interconnected via a 2D mesh network.

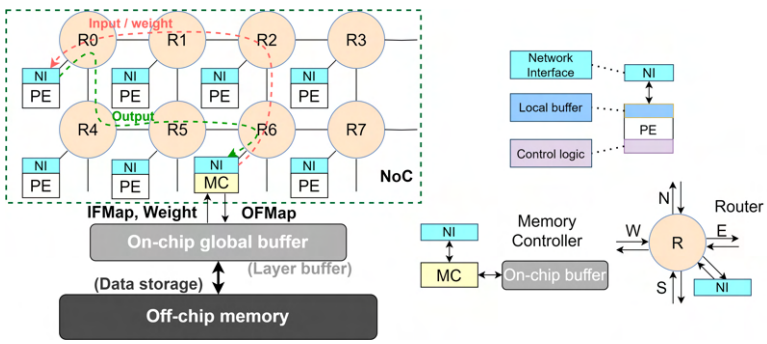


Figure 2.1: NoC-based DNN accelerator architecture.

NoC-based multi-core systems can employ heterogeneous cores to accommodate different tasks. In NoC-DNN, there are generally two types of cores. PE cores are used for neuron computations, which consist of the control logic, arithmetic logic units (ALUs), and local buffers. Memory controller (MC) cores control the

access to the memory for all data requests in the NoC. They also manage the task mapping and resource allocation for PE cores. The memory hierarchy for NoC-DNN is normally organized into two levels. The off-chip memory stores the weights and inputs for the whole DNN model, while the on-chip global buffer holds part of these data for fast fetch. Every processing core is connected to a NoC router through the network interface (NI). NI handles the data packetization, and the router manages the packet communication in the network.

2.2.2 Features of NoC-DNN simulators

To support a more flexible and realistic NoC hardware configuration prior to hardware level RTL prototyping as well as to maintain cycle-accurate behavior, we design the flit level NoC-DNN simulator, NoCDAS. It can track resource usage by clock cycles on a flit basis.

Since the NoC architecture for NNest, Timeloop, and STONNE are rigid, we only compare the two Noxim-based simulators [36, 38] that support flexible NoC configurations with ours. For a typical NoC-DNN simulator that accepts various DNN models, the computing task of the neuron network is converted to generic neurons. Then these neurons are mapped to PEs iteratively to perform parallel computation, and NoC manages the data transmission. Though CNN-Noxim, DNNoC-sim, and our NoCDAS can all evaluate full DNN models in this manner, several features can influence the utility of the simulator. We summarize the main differences across four features in **Tab.** 2.1, including the PE size, task mapping, weight memory, and dataflow.

Table 2.1: Feature comparison for NoC-DNN Simulators

	CNN-Noxim	DNNoC-sim	NoCDAS(our)
Assumed PE size (in neurons)	Large $10^3 \sim 10^6$	Large $10^3 \sim 10^6$	Small $1 \sim 10$
Task mapping	Limited by NoC size	Flexible	Flexible
Weight memory	Inside PE (All stored before simulation)	Inside PE (Update per mapping iteration)	On-chip buffer (Update per mapping iteration)
Dataflow in NoC	PE to PE	PE to PE	PE to MC MC to PE

CNN-Noxim and DNNoC-sim use weight stationary computation flow, where weight memory is placed inside PE. They assume a large-size PE that can hold thousands to millions of neurons simultaneously. As a result, the local buffer should be big enough to store the corresponding inputs in one data transfer. CNN-Noxim stores all the required weights locally in PE, therefore it only involves the PE-to-

PE data flow for inter-layer forwarding results transmission. Due to this setting, its task mapping is limited by the NoC size. A large DNN model can not be mapped to a small NoC when one of its layers cannot be fitted to the available PEs in the network at once. DNNoC-sim expands from CNN-Noxim with an off-chip memory module. It manages the weight update of local weight memories per mapping iteration, which assumes direct data transfer between PE and off-chip memory. The network is not involved in this update. In addition, the intermediate neuron outputs will be stored in the off-chip memory if the neuron tasks from the same layer need more than one mapping iteration. Therefore DNNoC-sim supports flexible task mapping, but its in-NoC traffic is still PE to PE.

In contrast to these simulators, our NoCDAS adopts a more realistic NoC model that involves the MC cores for managing the data request and accessing the two-level memory hierarchy. Moreover, we assume a small PE that can hold one or multiple neurons at a time based on the assigned hardware resources. In this case, our simulator can also evaluate large-scale NoC-based DNN accelerators.

CNN-Noxim and DNNoC-sim use the unicast data transmission mode for PE to PE data flow. In recent NoC-based DNN accelerators [27, 28, 41], other data transmission modes, such as multicast and broadcast, are used to enable data sharing across multiple PEs. These transmission modes are beneficial for convolution operations that require the same weight kernel for computation in the same output channel. Multicast and broadcast can reduce the network traffic and optimize the data transfer. However, they may also cause the scalability issue if multicast messages are broken down into multiple unicast packets in conventional NoC [42] or use specific topology such as tree-based multicast [28, 37, 43].

In our design, we adopt the unicast mode in a mesh NoC to support the data flow between PE and MC. It is possible to integrate additional multicast and broadcast transmission modes into our simulator in future versions. This can enable a more comprehensive exploration of data reuse methods and enhance the capability of NoCDAS to simulate more real-world NoC-DNN chips.

2.2.3 Proposed NoC-DNN simulator framework

The proposed NoC-DNN simulator is designed as a cycle-accurate simulator for NoC-based DNN hardware accelerators. **Fig.** 2.2 shows the overview of the simulator framework with its three major modules, the configuration input, the simulation platform, and the statistical output. The NoC-DNN simulator takes a DNN inference task as the input workload and generates corresponding results after a full model simulation.

2.2.3.1 Configuration input

The configuration files describe the DNN workload and the NoC parameters for simulation. Thus, two configuration files are generated as the simulator input.

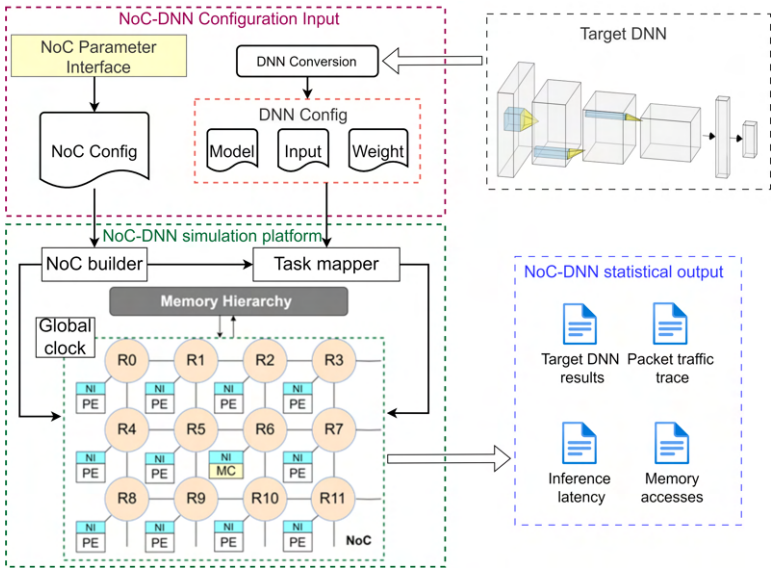


Figure 2.2: Overview of the proposed NoC-DNN simulator framework.

The first input file is the NoC configuration, which abstracts the target NoC hardware resources. A default parameter file (parameters.hpp) is provided for all adjustable variables, including the NoC size, core configurations, and a variety of network and router parameters. Since the simulated NoC is built according to the parameter macro definitions, the NoC hardware configuration changes require the simulator's recompilation before the evaluation.

The second is the DNN configuration, which contains information about the target DNN workloads. This configuration contains three files, including the DNN model structure, the input, and the weight. The model structure is described layer-by-layer, starting with the input layer. For each layer, the layer type, input size, filter size, and non-linear activation function should be provided. In addition, if any stride or padding is added, they should be specified in the model structure. To get accurate inference results, users need to prepare proper inputs and weights, preferably from a trained DNN model. As a lot of current DNN models are developed using open-source deep learning frameworks such as PyTorch [9], we provide a Python script to directly extract the weights from a pre-trained model that utilizes standard PyTorch APIs for DNN layers such as *nn.MaxPool2d* and *nn.Conv2d*. An example of preparing the weight and input files from a well-known CNN model LeNet-5 [44] using this script is given along with the simulator.

In addition to the full model execution (FE) mode, we provide a NoC-DNN hardware performance evaluation (RE) mode. This mode only involves the DNN model structure file as the DNN configuration and the simulator will run with random inputs and weights. Therefore, users can skip the complex training process,

facilitating faster evaluation.

2.2.3.2 Simulation platform

The simulation platform is the core of the proposed NoCDAS, which implements the hardware architecture shown in **Fig.** 2.2. The simulator is driven by a global clock, which makes the simulation process clock cycle accurate. Two interfaces connect to the configuration input module for building the NoC and mapping the DNN task to the NoC.

The first interface NoC builder constructs the NoC from the NoC configuration. It defines the cores, NIs, and routers and then places them in the network according to their position. After that, it links these NoC components together with the memory hierarchy to implement the NoC-DNN hardware. Simultaneously, the second interface task mapper converts and maps the target DNN onto the established NoC by parsing the DNN configuration.

Then we initialize the off-chip memory with given inputs and weights and reset the global clock. After tasks are distributed to PEs, NoCDAS starts the simulation. At each clock cycle, it will check the NoC-DNN execution status, run the *runonestep()* function for all PEs and MCs, and update the network traffic accordingly. Since the inter-layer data dependency exists in DNN, the simulation follows a layer-completing order to iteratively map the task of the next layer after the current layer is completed. In this case, there is no extra stall while awaiting requested data from the previous layer before they are updated in the on-chip buffer. The global clock drives the simulation and stops when all the DNN tasks are finished. The NoC platform is modified from the cycle-accurate NoC simulator in [45], which has a similar architecture to the widely used Garnet in Gem5 [46].

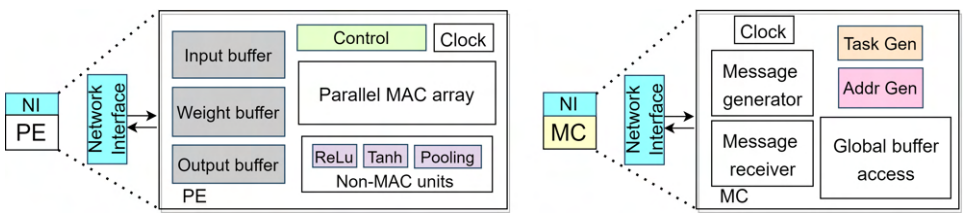


Figure 2.3: Basic modeling of PE and MC.

Fig. 2.3 illustrates the modeling of PE and MC cores. Since NI is modeled with packetizing and de-packetizing components [47], PE and MC will only process messages instead of NoC-level packets. PE consists of buffers and computational units. The parallel MAC array is for common MAC operations in DNN layers, it takes input and weight from the local buffers and generates the result. The non-MAC units are in charge of other operations such as non-linear activation and max pooling. The output buffer stores any intermediate result, and PE will create a message to carry the final result back to MC. MC consists of two message blocks

and a global buffer access block. The message receiver decodes the message from PE. Similarly, the message generator creates the response message to PE according to the request. The task and address generators are functional blocks designed to prepare neuron tasks and data addresses. Then MC can read and write the memory hierarchy through the global buffer access block.

2.2.3.3 Statistical output

Our simulator provides four output measurements to assess the performance of the user-defined NoC-DNN, including the DNN inference result, latency, traffic trace, and memory access.

1. Inference result: It shows the final output for full model evaluation, which signifies the efficacy and accuracy of the NoC-DNN simulation.
2. Inference latency: It outputs the end-to-end and layer-wised latency in clock cycles after simulation. This provides insights into the processing capability with the given NoC-DNN architecture.
3. Traffic trace: The packet traffic trace file records the NoC communications, which helps identify potential bottlenecks. This provides the potential for other trace-based NoC simulation tools to reproduce the DNN inference task without running actual inference workloads.
4. Memory access: This refers to the number of packets sent to and received from MC, which can be extracted from the trace file. It quantifies the utilization of the on-chip buffer.

These four statistical outputs contribute to a holistic assessment of the user-defined NoC-DNN accelerator, which helps perform statistical comparisons between different NoC-DNN configurations.

2.3 NoCDAS simulation workflow

Based on the proposed NoCDAS implementation, the simulator requires three steps to evaluate the target DNN model and NoC hardware specification. **Fig.** 2.4 shows the simulation workflow to get the final inference performance results.

2.3.1 DNN conversion process

The target DNN model usually comprises various types of layers, such as convolutional layers, pooling layers, and fully connected layers. In a fully connected layer (FC), one output is calculated with the MAC operations on all input neurons. For the pooling layer (Pool) and convolutional layer (Conv), one output only uses part of the inputs as a partially connected layer, whose size equals the shape of the pooling and convolution kernel, respectively.

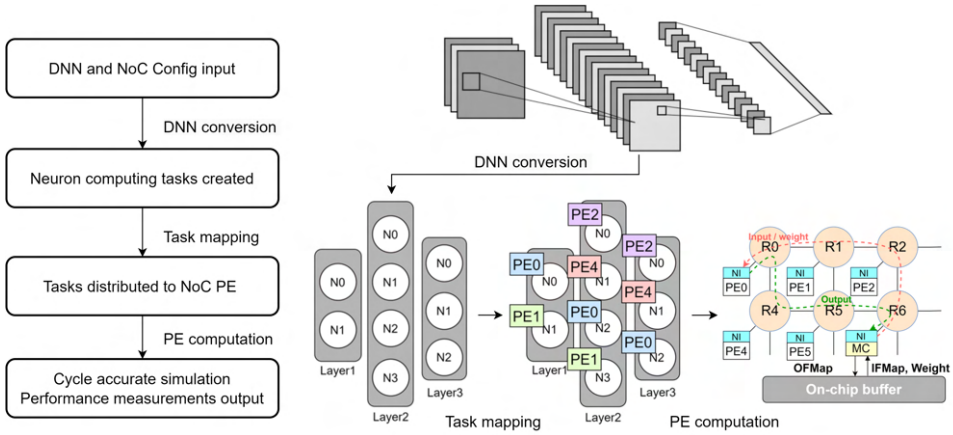


Figure 2.4: NoC-DNN simulation flow.

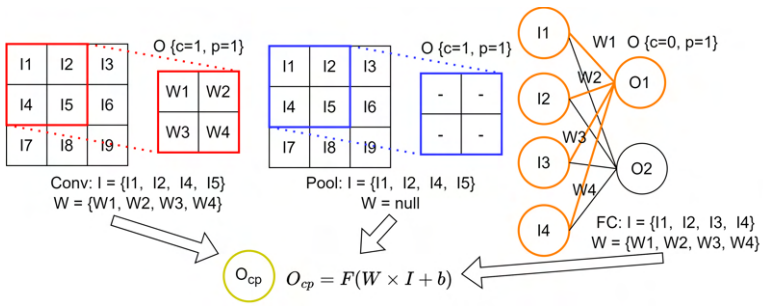


Figure 2.5: DNN conversion process for different layers.

Though the layer types are different, their basic neuron computing expression is similar to the formula $O_{cp} = F(W \cdot I + b)$ shown in **Fig.** 2.5. In which W and I are the weight vector and input vector, b is the bias, and F is the non-linear activation function. O_{cp} represents the output value of the neuron N_{cp}, where c means its channel if applicable and p means its position in the output feature map (OFMap). For FC and Conv layers, the size of vectors W and I are adjusted according to the layer and filter size. For Pool, a special F is adopted to fulfill the pooling operations such as Max and $Average$ instead of the normal non-linear functions. In this case, W is a null vector, and the output is calculated directly by $F(I)$.

Given a DNN model, we need to break down its entire workload into smaller tasks to distribute them across PEs in the NoC. To standardize these tasks, we convert DNN models to the granularity of individual neurons. The task of each neuron is to get its output no matter what layer operation it has. As a result, our simulator can run any DNNs that are built with standard convolutional, dense,

and pooling layers. During the DNN conversion process, layers in the DNN model are separated into neuron tasks, which ensure efficient parallelization over NoC hardware resources. The DNN conversion function is integrated with the task mapper interface.

2.3.2 Task mapping process

After the DNN conversion process, the created neuron tasks should be distributed to PEs in NoC. Since the number of PEs is limited compared with the converted neurons, especially for large-scale DNN models, we need a task mapping process to ensure the correct execution of these tasks.

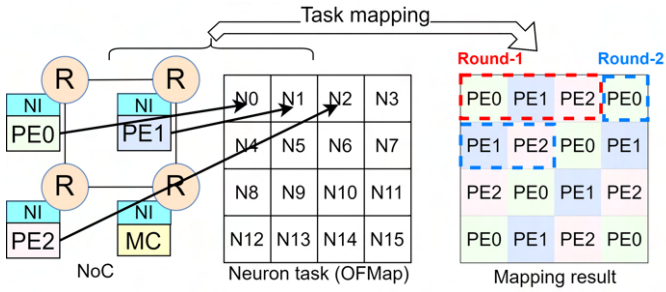


Figure 2.6: Task mapping process over PE array.

The simulation follows a layer-completing order, so we separate the neurons layer by layer, and the task mapping process for each layer is called a mapping iteration. The basic mapping strategy considers a balanced workload distribution of neuron tasks over the whole NoC. The prevalent method is even mapping in Eyeriss V2 and CNN-Noxim [29, 38]. We adopt this method as it can be applied to any DNN tasks run on NoCDAS. Here we assign neuron tasks to PEs in the order of row-oriented mapping, until all neurons in this layer are attached to one of the PEs. **Fig.** 2.6 gives a mapping example of a 4×4 OFMap over three PEs. In this example, we assume at each time one PE can execute one neuron, while other mapped tasks are waiting in line. This is similar to the Direct-X mapping in CNN-NoXim but the scale of each mapped task is much smaller. Therefore, we don't need to allocate enormous local buffers inside each PE for the multi-core NoC.

The task mapper will create a mapping table for each layer, and store the mapped task IDs as a task list in its corresponding PE before the layer execution. Then PE knows the exact tasks it should perform during this mapping iteration. The task list updates after the layer is completed. Simultaneously, the on-chip global buffer updates the stored weights from the off-chip DRAM for the next layer.

Note that we support flexible configuration on mapping strategy to evaluate various data flow control mechanisms on the same NoC. This will change the task list in PEs, thus affecting the execution order of the converted DNN. In addition, the PE capability can be customized to process multiple neurons at a time to simulate some powerful and heavy PE designs.

2.3.3 PE computation flow

After the task mapping process, each PE has its dedicated neuron tasks to compute. We define a computation round as the time for processing one neuron. The proposed NoCDAS requires multiple rounds to complete the computation in one mapping iteration (or one layer). Since we need to simulate the network traffic accurately, the PE computation flow of one neuron is formulated in data packets. In each round, the PE activity to compute one neuron output can be divided into three steps:

(a) In the first step, PE sends a data request packet to MC, which involves the neuron task ID to fetch weight vector W, input vector I, bias b, and also the indicator for layer function F from the global buffer.

(b) In the second step, MC sends the requested data packet back to PE. MC accesses the on-chip buffer and prepares the data. The packet length depends on the amount of data as its payload. PE will start the computation after all the required information is received.

(c) In the third step, PE sends the result output packet to MC after its task is finished. The NoC configuration parameters define the computation latency of PE.

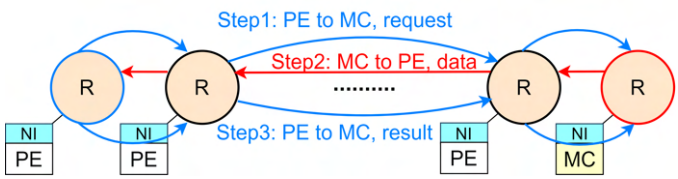


Figure 2.7: PE computation flow in three steps.

Fig. 2.7 shows the traffic trace of these three steps between a PE and an MC. The destination MC of each PE also depends on the mapping strategy. In NoC-DNN, multiple PEs and MCs work on a large DNN model, generating complex traffic patterns. As a result, the proposed flit-level simulator NoCDAS can handle and record the traffic accurately for further evaluation.

In addition to the current PE computation flow, the NoCDAS simulator is not restricted to the small PE model. Users can attach more than one PE to the router

to simulate a larger PE cluster that can process multiple neurons in one data transfer. Meanwhile, the task mapping and PE computation flow need adaptation to fit the large PE scenario.

2.4 Evaluation and exploration showcase

2.4.1 Experiment setup

To evaluate the proposed NoCDAS, we run three well-known DNN models on it, which are a small model LeNet [44], a medium model AlexNet [5] and a large model DarkNet [48]. Two groups of experiments are designed, including validating the simulator functionality and exploring NoC hardware configuration on running DNN models. The validation experiments will first examine the inference results of DNN with full model evaluation. Then the correctness of the hardware performance evaluation mode is validated by the inference latency using the same NoC configuration. During validation, we will use LeNet and AlexNet as our targets. After the functionality validation, we will explore various NoC-DNN design parameters and their affection on DNN inference latency using all three DNN models.

As a default setup, we initialize an 8×8 mesh NoC via NoC configuration. The NoC configuration parameters and their default values are summarized in **Tab.** 2.2. The network uses Virtual Channel (VC) routers, where several channels share a single physical channel to use the bandwidth efficiently. The protocol for transmission between VC routers involves flit-based wormhole transmission combined with credit-based flow control [24]. One data packet is divided into smaller units called flits for fine-grained resource allocation. Each upstream router keeps track of the number of free flit buffers in each virtual channel downstream using a credit mechanism. Credits are decremented when the upstream router sends a flit and incremented when the downstream router frees the associated buffer. This ensures that the downstream router has sufficient virtual channel buffer space before flits are transmitted, preventing buffer overflow and thus no data loss.

We adopt a deterministic X-Y routing algorithm to avoid deadlock. The router frequency depends on the complexity of its logic for routing and resource allocation. In canonical 2D mesh-based multi-core System-on-Chip design, NoC can run much faster than processing cores [49]. We set the router frequency at 2 GHz as a common configuration [45], and the PE frequency is set to 200 MHz as in [50]. For the memory hierarchy, we simplify the memory access latency as MC read delay and MC data fetching delay. The latter is calculated by the bandwidth of the on-chip buffer. We set the MC transmission bandwidth to be 12.8 GB/s and the read latency is 5 ns (standard DDR3-1600 SDRAM) [51].

The default unit for latency measurement is one router clock cycle. For ease of simulation, we arrange router IDs in a row-major order starting with ID 0 and separate the whole network into four identical 4×4 building blocks. In total 8 MCs are deployed in the NoC, therefore 2 MCs are placed in each building block. **Fig.** 2.8 illustrates the building block on the left upper corner of the default NoC

Table 2.2: NoC configuration parameters (default)

Topology	Mesh	Size	8×8
Flow control	Virtual channel	Arbitration	Round-Robin
VC per port	4	VC buffer depth	4 flits
Routing protocol	X-Y routing	Link width	256 bits
Link direction	Bi-direction	Data width	16 bits
Link latency	2 cycles	Router latency	1 cycle
MC bandwidth	6.4 Byte/cycle	MC access latency	1 cycle
Number of PEs	56	Number of MCs	8
Router frequency	2 GHz	PE frequency	200 MHz
PE capability	25 OPs / 1 non-linear activation per PE cycle		
Task mapping	Row-major mapping		

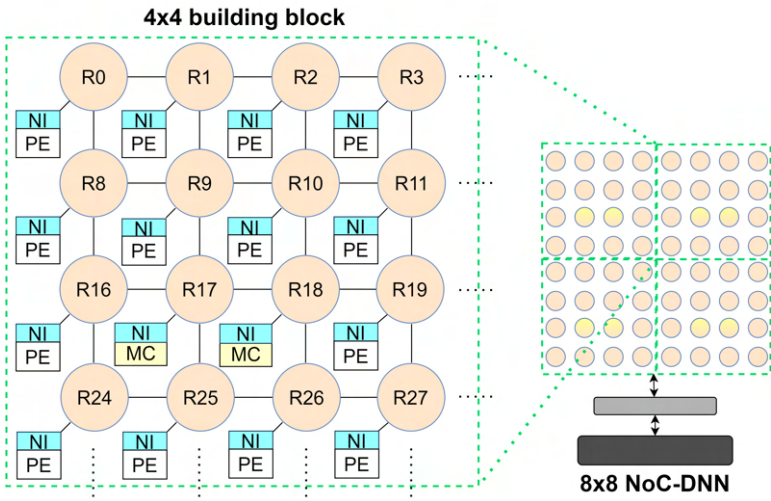


Figure 2.8: Default NoC structure and its building block.

configuration, where two routers with ID 17 and 18 are connected to MC. Each PE is assigned to the nearest MC within its block to minimize communication latency and balance the load across the MCs. This assignment is a design-time decision. In this default building block, PEs from the left two columns communicate with the MC at router R17 and the PEs from the right two columns communicate with the MC at router R18.

Based on the default configuration, users can adjust the parameters to simulate different NoC-DNNs. The NoC mesh size can be arbitrary positive integers. The VC number, VC buffer depth, link width, NoC structural latency, and frequency can be changed according to users' design. The MC placement can be customized. There are seven sets of NoC size and MC position predefined in the parameter

file in NoCDAS. Users can add more NoC size and MC position setups by editing the MAC class in the NoCDAS program. The destination MC for each PE should be assigned prior to simulation, which ensures correct traffic in NoC. We provide detailed guidance in the simulator README file.

The information of target DNN models is saved in DNN model structure files as part of the DNN configuration, and each line represents one layer in the model. The first line tells the width, height, and channel of the input layer. Then for Conv layers, the description includes the input channel, filter width, filter height, output channel, activation function, padding, and stride. For Pool layers, the parameters are the same as Conv layers, except for the activation function. For FC layers, only input size, output size, and activation function are required. The task mapper interface will perform DNN conversion and the converted parameters used in the simulation are listed in **Tab.** 2.3 (LeNet), **Tab.** 2.4 (AlexNet), and **Tab.** 2.5 (DarkNet). For DarkNet, we adopt the version with 19 convolutional layers, which is used as a backbone network for object detection model YOLOv2 [48]. The number of neurons counts the neuron tasks after DNN conversion. The number of OP counts the required operations for each layer, such as MAC operations for the convolutional layer, and Argmax comparisons for the max pooling layer. For example, the #OP in Conv1 layer of AlexNet is calculated by $(3\times11\times11)\times290400 \approx 105.4$M. For ease of simulation, we assume that each operation takes the same time, and the PE capability is 25 OPs or 1 non-linear activation per PE cycle, as shown in **Tab.** 2.2. The number of rounds is calculated by $\lceil$#Neuron/#PE$\rceil$, which counts the neuron tasks mapped to each PE on average during one mapping iteration. In **Tab.** 2.3, **Tab.** 2.4, and **Tab.** 2.5 we calculate the number of rounds under the default NoC configuration.

Table 2.3: LeNet parameters for simulation (Input size $32 \times 32 \times 1$)

Layer	#Neuron	#Round	#OP	ACT func.
Conv1	4704	84	117.6k	ReLU
Pool1	1176	21	4.7k	-
Conv2	1600	29	240k	ReLU
Pool2	400	8	1.6k	-
FC1	120	3	48k	ReLU
FC2	84	2	10.1k	ReLU
FC3	10	1	840	Sigmoid

The configuration parameters for exploration are specified in the corresponding experiment subsections below. The inference latency is recorded in router cycles.

2.4.2 Simulator validation

Firstly, we evaluate the inference results under full model execution (FE) mode to validate the proposed simulator NoCDAS. We use LeNet and AlexNet as the target

Table 2.4: AlexNet parameters for simulation (Input size $227 \times 227 \times 3$)

Layer	#Neuron	#Round	#OP
Conv1	290400	5186	105.4M
Pool1	69984	1250	0.63M
Conv2	186624	3333	447.9M
Pool2	43264	773	0.39M
Conv3	64896	1159	149.5M
Conv4	64896	1159	224.3M
Conv5	43264	773	149.5M
Pool3	9216	165	0.08M
FC1	4096	74	37.7M
FC2	4096	74	16.8M
FC3	10	1	409.6k

Table 2.5: DarkNet parameters for simulation (Input size $256 \times 256 \times 3$)

Layer	#Neuron	#Round	#OP	Layer	#Neuron	#Round	#OP
Conv1	2097152	37450	56.6M	Conv9	131072	2341	302M
Pool1	524288	9363	2.1M	Conv10	65536	1171	302M
Conv2	1048576	18725	302M	Conv11	131072	2341	302M
Pool2	262144	4682	1.05M	Conv12	65536	1171	302M
Conv3	524288	9363	302M	Conv13	131072	2341	302M
Conv4	262144	4682	302M	Pool5	32768	586	0.13M
Conv5	524288	9363	302M	Conv14	65536	1171	302M
Pool3	131072	2341	0.52M	Conv15	32768	586	302M
Conv6	262144	4682	302M	Conv16	65536	1171	302M
Conv7	131072	2341	302M	Conv17	32768	586	302M
Conv8	262144	4682	302M	Conv18	65536	1171	302M
Pool4	65536	1171	0.26M	Conv19	64000	1143	65.5M

DNN models. The input images are randomly chosen from MNIST for LeNet and resized CIFAR10 for AlexNet. The weights are trained in PyTorch and extracted by our script. The output feature map of each layer and the final inference results are printed from NoCDAS. Then we compare them with the PyTorch version and get the identical layer outputs and classification values. The correctness of NoCDAS can be ensured since the NoC packet delivery is correctly executed to get all the DNN neuron tasks done as we designed.

Then we validate our simulator's performance evaluation (RE) mode, which only takes the DNN structure file as DNN configuration. In RE mode, the weights and inputs are randomly generated for a quicker simulation flow without the DNN training process. Thus the inference results are not comparable for validation. In this case, we use the layer-wise latency results to validate that RE mode can give

the same performance output as FE mode. We run the same DNN models with the default NoC setup in NoCDAS.

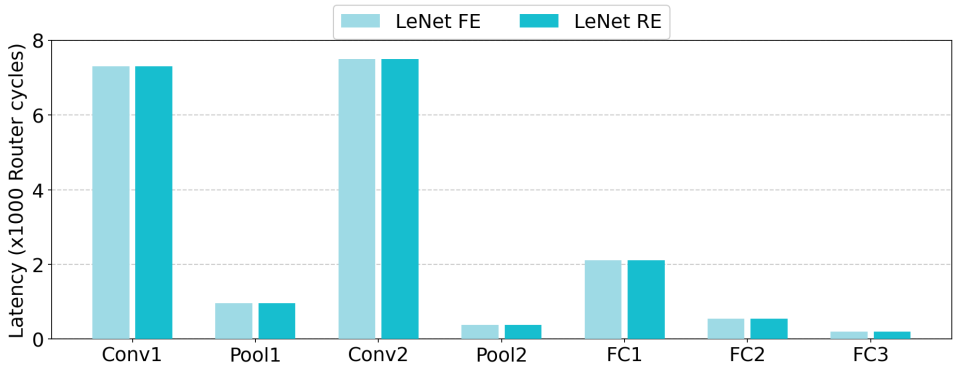


Figure 2.9: LeNet layer-wise latency comparison between FE mode and RE mode.

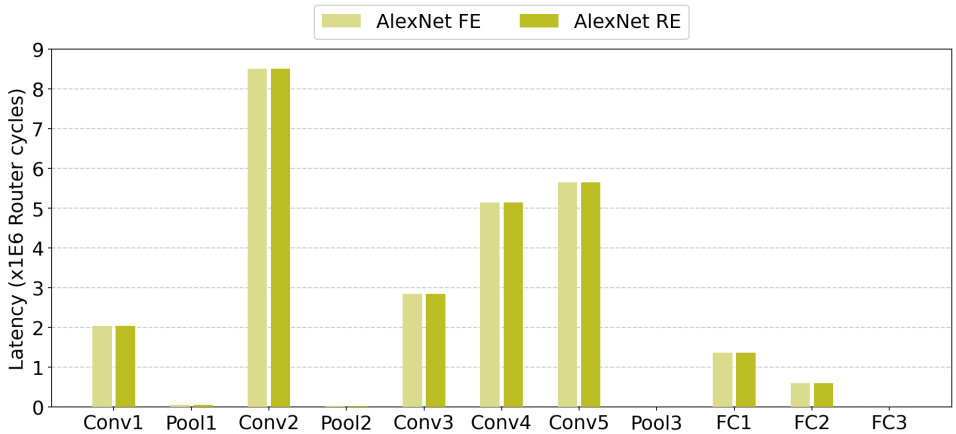


Figure 2.10: AlexNet layer-wise latency comparison between FE mode and RE mode.

Fig. 2.9 and **Fig.** 2.10 plot the layer-wise latency of LeNet and AlexNet of two evaluation modes, in which we can observe identical inference latency in both modes. Thus the correctness of hardware performance measurement in RE mode is validated. A traffic trace file is generated along with the latency result, which records the packet ID, packet length, source, destination, creation time, and elimination time of every packet in NoC. There are 24282 packets created for LeNet and 2342238 packets created for AlexNet. Both of them are identical to the theoretical number, which is three times the number of neuron tasks after DNN conversion. The memory accesses are also extracted from this trace file.

Besides the function validation, we also measure the real execution time of our simulator. We execute all three DNNs under RE mode using the default configuration as listed in **Tab.** 2.2. We run the simulator on a desktop computer with Intel Core i5-10600 CPU and 16GB RAM. The runtime of LeNet is 1.195 seconds. AlexNet and DarkNet need 3258.11 seconds and 10098.528 seconds to complete, respectively. The average runtime per simulator clock cycle is around 0.077 milliseconds.

2.4.3 NoC-DNN design parameter exploration showcases

We present the NoC hardware exploration showcases upon the validation using the proposed NoCDAS. There are large design spaces for the NoC-DNN architectures, here we select three common design parameters as targets, including the NoC size, task mapping, and MC placement.

1. NoC size: It defines the network size in NoC-DNN, which represents the scalability of the network. In the default setup, we use an 8×8 network with 4 building blocks, now we extend it to 12×12 and 16×16. In this case, they consist of 9 and 16 building blocks. We also evaluate it on a small NoC with only one 4×4 building block.

2. Task mapping: It defines the neuron task mapping strategies. The default mapping is row-major mapping, which maps the neuron tasks along each row of PE. It is similar to the Direct-X mapping method [38] in CNN-Noxim. We also test two other mapping methods, column-major and random mapping. **Fig.** 2.11 gives the example task mapping outputs 16 neuron tasks on three PEs. The neuron task execution sequence in PE depends on the mapping result.

3. MC placement: It defines the amount and position of MC in NoC. This will affect the data transmission distance between PE and MC, and change the NoC activity. In addition to the baseline setup, two new building blocks are designed for evaluation. The first design places 2 MCs on the edge of the block. The second design only reserves one MC at the center of the 4×4 block.

2.4.4 NoC-DNN design parameter exploration results

Fig. 2.12, **Fig.** 2.13, and **Fig.** 2.14 draw the inference latency results of three DNN models with two groups of design parameters, including four NoC sizes and three task mapping methods.

We first discuss the impact of NoC size using row-major mapping. With the quadratic growth of processing cores in NoC, the overall inference latency decreases exponentially in all cases. Since the NoC is configured to use the same building block

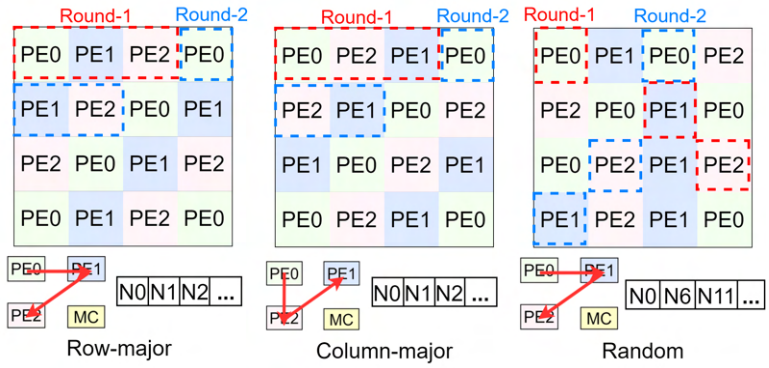


Figure 2.11: Task mapping examples.

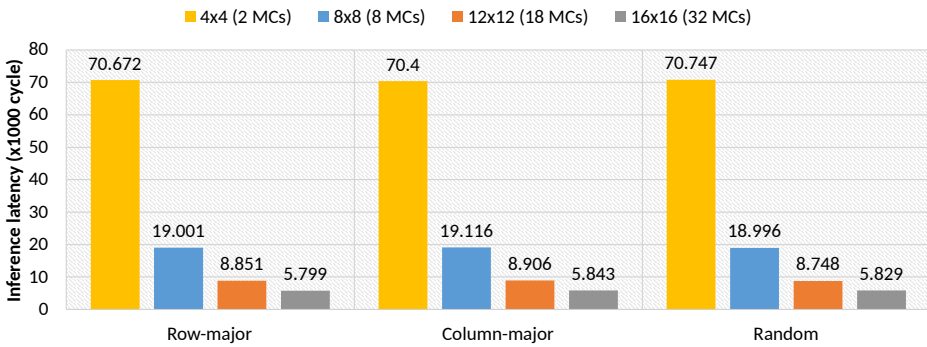


Figure 2.12: LeNet inference latency. Design parameters: NoC size and task mapping.

structure, we can investigate the scalability efficiency by comparing the equivalent latency over one building block (1U), E_{lat} = Latency $\times$ #**U**. We normalize each E_{lat} based on the default 8×8 NoC. A smaller E_{lat} represents a better scalability efficiency on the target DNN.

Table 2.6: The equivalent latency of different NoC sizes, normalized according to the default 8x8 NoC (4U).

E_{lat}	4x4 (1U)	8x8 (4U)	12x12 (9U)	16x16 (16U)
LeNet	0.8256	1	1.0481	1.2208
AlexNet	3.0353	1	0.8313	0.8378
DarkNet	1.1560	1	0.9563	1.0487

All equivalent latency results are shown in **Tab.** 2.6. For LeNet, the 4×4

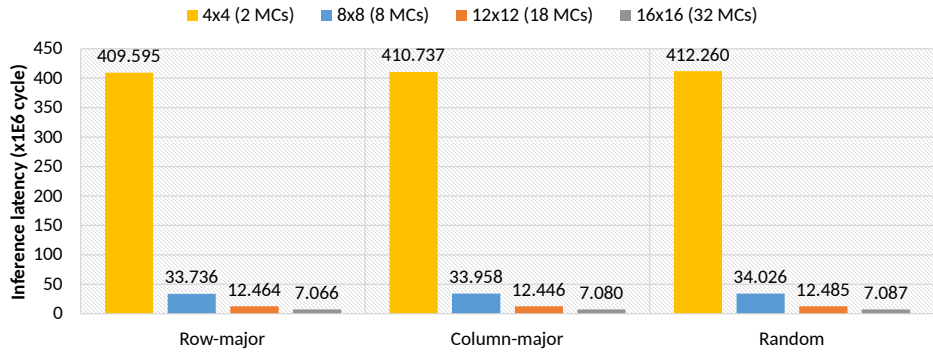


Figure 2.13: AlexNet inference latency. Design parameters: NoC size and task mapping.

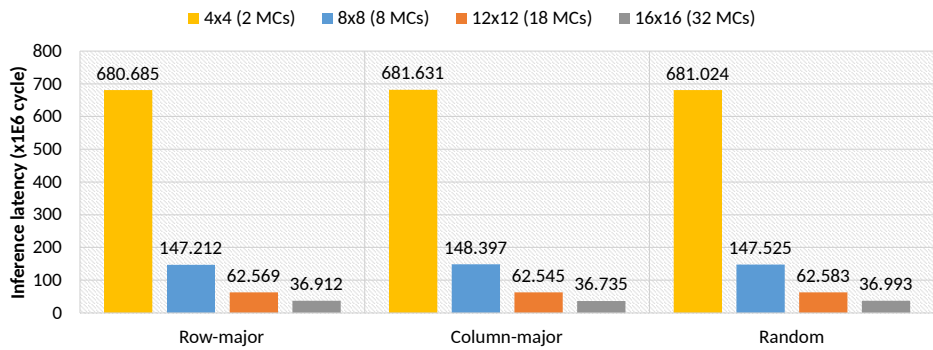


Figure 2.14: DarkNet inference latency. Design parameters: NoC size and task mapping.

setup has the smallest E_{lat} over four NoC sizes. Since the LeNet doesn't have a tremendous number of tasks as modern DNNs, a small-sized NoC-DNN can efficiently fit such a workload. For the medium DNN model AlexNet, the 12×12 NoC reaches the best scalability efficiency, and the E_{lat} of 16-by-16 is on the same level with around 0.6% difference. For the large DarkNet model. the 12×12 NoC has the best E_{lat} over four setups, and we can observe that 16×16 gives a worse E_{lat} compared with the default NoC. Therefore, a bigger NoC-DNN doesn't mean higher scalability efficiency. The proposed NoCDAS simulator can quickly find the most efficient scale of NoC to execute the target DNN models. In our showcase of NoC size exploration, a 4×4 NoC is suitable for LeNet and a 12-by-12 NoC can be chosen for AlexNet and DarkNet.

Then we evaluate the impact of task mapping strategies in NoC-DNN. **Fig.** 2.15 shows the normalized inference latency of three mapping methods over five NoC

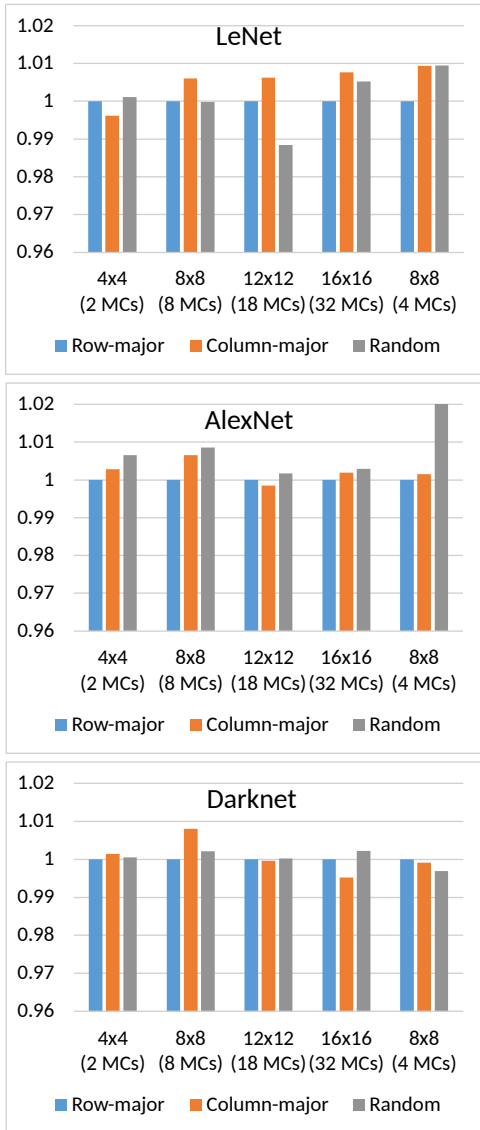


Figure 2.15: Inference latency comparison of three task mapping methods, normalized to row-major mapping.

configurations based on the row-major mapping results. We can observe that the differences are not obvious among them. Since the computation tasks are designed to be evenly distributed to all available PEs in the evaluated mapping methods, PEs have the same amount of neuron tasks for each mapping iteration. Thus they have similar end-to-end latency. The differences happen mainly in the tail round, where some of the PEs are idle since the remaining tasks are less than the number of available PEs. In this case, the occupied PE will change according to task mapping, which would affect the latency in that round. For LeNet, row-major and column-major mapping are similar in all cases, since the workload distribution is similar in their tail round. For random mapping, only the 12×12 NoC shows a larger gap, which happens mainly in layer Conv2. The possible reason is that the 88 neuron tasks in this tail round got a better PE positioning in NoC which shortened the data path. For AlexNet, the 8×8 (4 MC) NoC has the largest gap between the random mapping and the other two mappings. This happens mostly because of Conv2 and Conv3 layers as they have medium-sized tail rounds, where the differences in task mapping can be exposed. For DarkNet, the latency among different mapping is not obvious. Since the number of rounds in each convolution layer is huge, the latency difference of the tail round is concealed.

The neuron workloads of each PE using the three mapping strategies in this showcase are balanced. More complex mapping algorithms that consider uneven neuron task distribution have the potential to optimize the inference latency further. NoCDAS can be used to evaluate them efficiently.

Finally, the MC placement is evaluated by three different building blocks with the same NoC size of 8×8. Besides the default MC placement (8 MCs, center), we use two alternative designs. The first design (8 MCs, edge) still employs 2 MCs in each building block, but we place them on the edge. Similar to the example in **Fig.** 2.8, routers R8 and R16 are connected to the MC in this case. PEs in the top two rows are assigned to MC at router R8, and those in the bottom two rows are assigned to MC at router R16. The second design (4 MCs) uses the single-MC building block, where the router R18 is connected to the MC. With this change, the data transmission inside the building block becomes all-to-one traffic. The single-MC placement will worsen the performance compared to the other dual-MC scenarios due to the heavier MC accesses and longer communication latency

Fig. 2.16 plots the inference latency of the DNN targets under these three MC placements. Though the 4-MC placement has more PE cores, it requires on average 1.18×, 1.90×, and 1.35× time to execute LeNet, AlexNet, and DarkNet compared with the 8-MC placements. We can observe that MC becomes the bottleneck that limits the processing efficiency in such NoC-DNN configurations. Considering the two 8-MC cases, the design of edge placement needs 1.12×, 1.42×, and 1.10× inference latency on DNN models compared to the design with MCs placed at the center. This is mainly caused by the enlarged communication distance, with an average of 2.3 hops between PEs and MCs in the 8-MC edge placement case, and 1.7 hops compared to 1.7 hops in the default placement case. By varying the MC placements, NoCDAS provides a rapid exploration of PE and MC resource

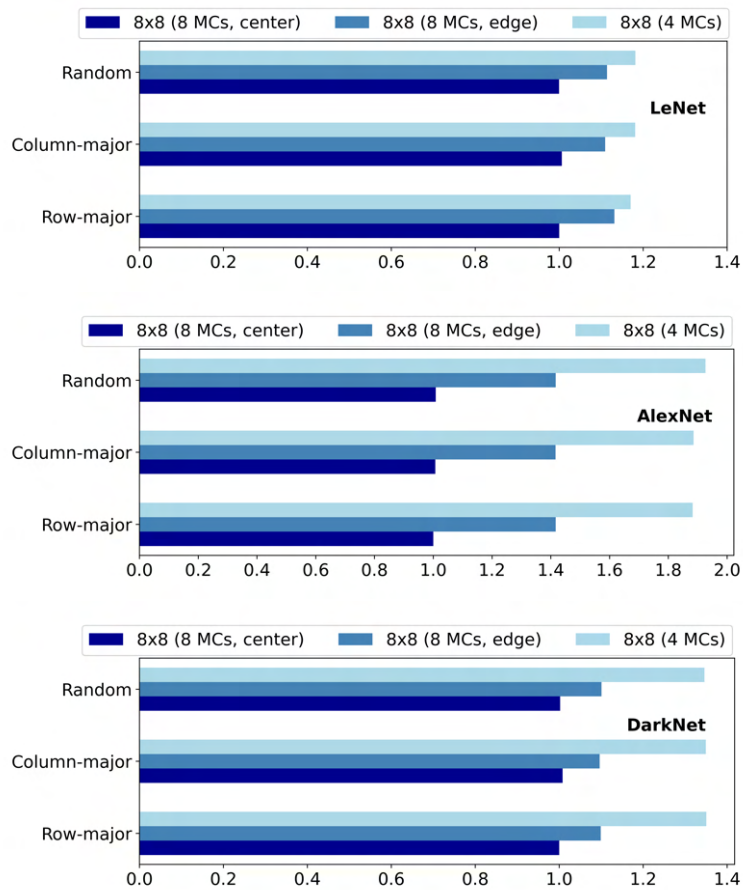


Figure 2.16: LeNet, AlexNet, and DarkNet inference latency, normalized according to the default setup. Design parameter: MC placement.

allocation.

2.4.5 Comparison with DNNoC-sim

In this section, we compare the proposed NoCDAS with the DNNoC-sim to show the design parameter exploration ability differences between the two simulators. Since the computation flow and hardware assumption are different in the two simulators, the latency result is not directly comparable. Hence we evaluate them regarding the scalability exploration. By changing the NoC size parameter, one can analyze the simulator output to deploy the proper amount of NoC hardware resources for a certain DNN target.

For DNNoC-sim, we adopt the LeNet and VGG16 inference latency results on mesh-based NoCs from [36], which represents the small and large DNN workloads. We transform the time unit from second to router cycle according to their system frequency setup. Since different group sizes for partitioning the DNN models are reported for DNNoC-sim, we select the smallest one in both scenarios, which are 216 neurons per group for LeNet and 200 thousand neurons per group for VGG16. For NoCDAS, we change the NoC size and keep other NoC configuration parameters in default to run the same DNN models.

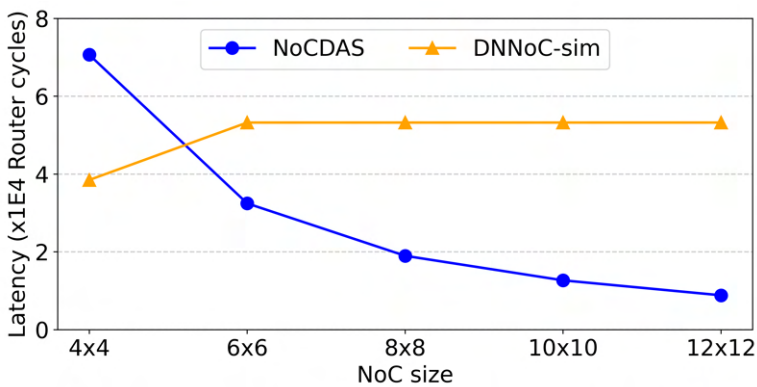


Figure 2.17: LeNet end-to-end inference latency when varying NoC sizes in DNNoC-sim and NoCDAS.

Fig. 2.17 and **Fig.** 2.19 plot the inference latency results of LeNet and VGG16 in two simulators, where the evaluated five NoC sizes range from 4×4 to 12×12. Note that for 6×6 and 10×10 NoC, the default building block in NoCDAS cannot directly cover their area, so we assign 5 and 13 MCs respectively to keep the proportion of MC according to NoC size. In addition, the latency breakdown of LeNet and VGG16 are plotted separately in **Fig.** 2.18 and **Fig.** 2.20. For NoCDAS, the computational latency records the calculation time in PE. The data transmission latency involves the NoC communication time and the memory access time.

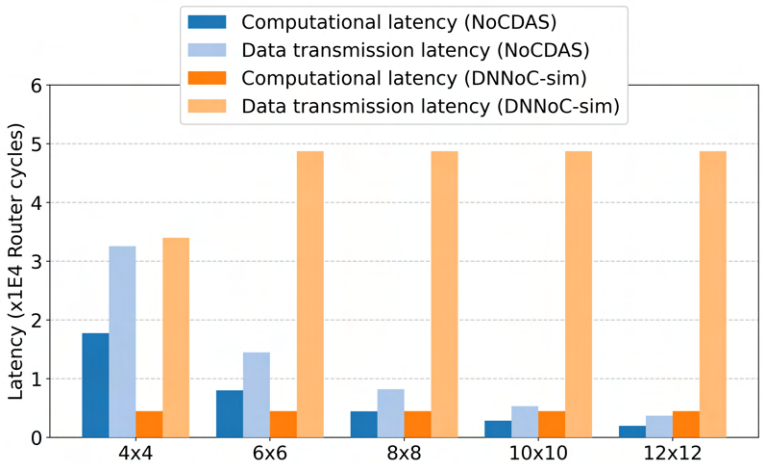


Figure 2.18: LeNet inference latency breakdown when varying NoC sizes in DNNoC-sim and NoCDAS.

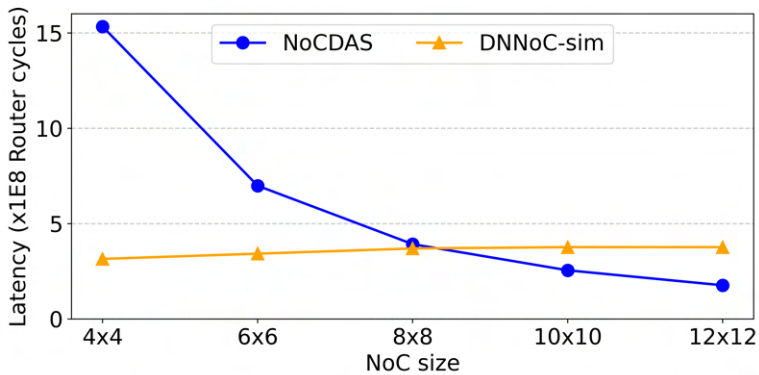


Figure 2.19: VGG16 end-to-end inference latency when varying NoC sizes in DNNoC-sim and NoCDAS.

Since the neuron tasks are evenly mapped over all PEs in hundreds of computation rounds, we count the latency on each neuron task and then average among all PEs in NoC to represent the actual workload on each PE.

For DNNoC-sim, the memory access latency is not included in its performance evaluation results, therefore we exclude this part from the shown data transmission latency. The computational latency in DNNoC-sim analysis is the sum of the PE computation time for each layer in each mapping iteration. Under their group setting, the LeNet model can be finished in two mapping iterations for the 4×4 NoC, and only one mapping iteration for other NoC sizes. For the VGG16 model,

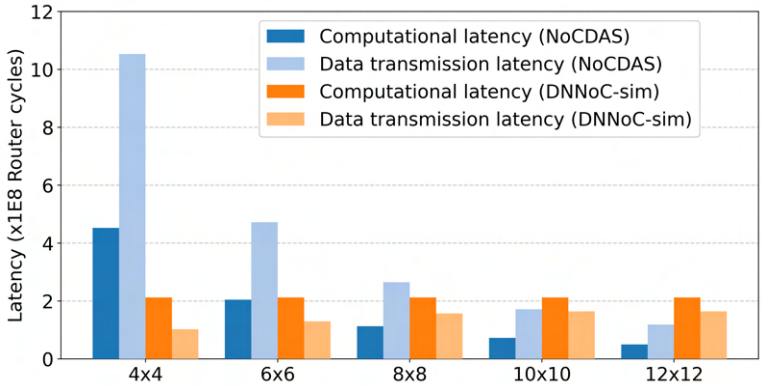


Figure 2.20: VGG16 inference latency breakdown when varying NoC sizes in DNNoC-sim and NoCDAS.

though it requires five mapping iterations for the 4×4 NoC, the largest layer can still be mapped to this NoC at once. Therefore, their computational latency remains the same when varying the NoC size.

In NoCDAS, the inference latency is inversely proportional to the NoC size, which indicates the better parallelism of increased processing cores in the network. However, in DNNoC-sim, the results are different. For LeNet, the inference latency first increases and then becomes saturated after the NoC size reaches 6×6. For VGG16, the inference latency increases when the NoC size becomes larger and saturates at 10×10. The possible reason is the limited network utilization, which arises from the impractical assumption of large PE size and the ignored communication overhead from PE to memory in DNNoC-sim. When the neuron tasks in adjacent layers cannot be mapped simultaneously, the temporary OFMap results are written directly to the memory by PEs and later read by the corresponding PEs in the next mapping iteration. There is less traffic in the NoC in this situation. Therefore the data transmission time in LeNet is shorter in the 4×4 NoC as it's the only setup unable to map LeNet onto the NoC in a single step. Similarly, for VGG16, the data transmission latency increases until the 10×10 NoC, as the two largest NoC sizes in our experiment can map VGG16 in one mapping iteration.

As NoC-DNN is designed to achieve parallel processing in PEs, the performance should be improved when the scale of NoC increases. However, the increased size of NoCs leads to poorer overall latency results in DNNoC-sim, which negatively impacts its reliability. In contrast, the proposed NoCDAS can well simulate such scalability in NoC-DNN, therefore giving designers the correct insight into the design parameter of accelerator size.

2.5 Summary

The presented experimental results validate the correctness of our simulator NoC-DAS and exhibit its unique ability on NoC-DNN design parameters exploration over other NoC-DNN simulators. We showcase the NoC-DNN design exploration over three design parameters, including NoC size, task mapping, and MC placement. Three DNN models encompassing LeNet, AlexNet, and DarkNet are tested to represent light to heavy workloads on NoC-DNN. Through these demonstrations, the potential of our simulator to guide on selecting latency and cost-efficient NoC-DNNs and assist better NoC-DNN hardware designs is proved.

Future work may include the evaluation of different optimization strategies and workloads as add-ons in our simulator, such as layer-pipelining, in-network processing, sparse DNNs, and transformers. In-depth explorations of design parameters can help improve the efficiency of NoC-DNN. Multicast and broadcast transmission can be added for the exploration of data reuse methods. More performance measurements like the power model can be integrated into our simulator. The parallelization of simulation is also important to reduce running time when simulating large-scale NoC-DNNs.

Chapter 3

Efficient NoC-DNN Architectures

3.1 Introduction to in-network processing

Modern Deep Neural Network (DNN) models usually contain millions of parameters and run billions of computations for a single inference process, therefore hardware accelerators are designed to reduce its latency. As each neuron performs simple multiply-accumulation (MAC) operations in a large amount, most DNN accelerators employ hundreds of processing elements (PEs) [8] in parallel to increase throughput.

NoC-DNN has recently been an active research area. The majority of such implementations perform careful co-design with PE and NoC to optimize data traffic aiming for specific DNN layer structures [7]. In this case, most efforts on NoC-DNN have been put into topology designs to support flexible communication, and reduce throughput and power/energy overheads [28, 29, 41, 52]. On the other hand, their performance is still not optimized from the NoC perspective due to the data communication delay, especially in the network congestion situation.

With the inspiration of in-network processing [53], some of the necessary operations in DNNs, such as non-linear activation functions, can be conducted in network routers with minimal influence on transmission delay to decrease the overall latency. In this paper, we propose a computation-while-blocking technique to conduct activation in network (AiN) that can be seamlessly integrated with current NoC-based DNN accelerator designs to improve the inference speed. We modify the typical NoC virtual channel (VC) router architecture to offload the non-linear activation function from PEs to routers. We evaluate our technique on a cycle-accurate NoC-DNN simulator and discuss the possible performance improvement for different DNN layers.

This chapter is based on the material from papers 2 and 3.

3.2 Related work

NoC-based hardware accelerators are common for general-purpose deep learning applications with myriad layer configurations [7]. Recent NoC-DNN works have exerted great effort into energy efficiency optimization. Zhu *et al.* implement a NoC-based DNN accelerator with the sparse skipping technique to reduce memory access overhead [52]. Other researchers focus on increasing computing flexibility and throughput for different traffic patterns used in NoC-DNNs. Multiply-Accumulate Engine with Reconfigurable Interconnect (MAERI) [28] employs tree-based topology to support various DNN partitions and mappings via tiny configurable switches. However, as the tree topology increases the time complexity for data communication, the scalability of MAERI is not as good as mesh-based NoC. In [41], a reconfigurable mesh-based NoC-DNN accelerator is proposed to cover different kernel sizes for convolution operation. It also adopts the channel-based weight-wise convolution and input-oriented approach to reuse the input and weight, which reduces memory access and improves throughput.

3.2.1 In-network computation for NoC-DNN

These hardware designs have mainly focused on multiply-accumulation (MAC) operations in the spatial PE array architecture. Other operations, such as the non-linear activation, are overlooked though they appear as an important part of the DNN inference process. These operations are simply attached in PE in their optimized NoCs. The concept of in-network processing [53] points out the feasibility of offloading such computations to the network to reduce the end-to-end latency. Lu presents in-network packet generation to engage NoC actively in computation to reduce lock coherency overhead [53]. Zheng *et al.* develop IIsy for in-network classification using off-the-shelf programmable switches [54]. It can run small models such as Random Forest purely inside the network without any back-end cores. But for DNNs, it is not applicable.

In contrast to the above works, we propose a new technique to improve the NoC-based DNN accelerator design by offloading part of the DNN calculations from PEs to routers, specifically the non-linear activation operations. This approach can be applied to NoC-based DNN accelerators to shorten the inference latency.

3.2.2 Pooling layer processing in NoC-DNN

Recent works regarding NoC-CNNs employ novel NoC structure designs to reduce energy consumption in computational-heavy layers [27, 28]. Other researches on NoC-CNN improve hardware utilization for convolution layers [36, 55]. However, these state-of-the-art optimization techniques often bypass pooling layers, though they are essential in CNN models to reduce spatial dimensions.

Two common approaches exist for processing pooling layers. The first is to treat pooling as a standalone layer, as adopted by the aforementioned NoC-CNN

architectures [27, 28, 36, 55]. They use local or global buffers to store intermediate layer outputs and conduct inference in a layer-completing order. This approach reduces design complexity and offers adaptability to various workloads. However, it introduces overhead in communication and memory access for distributing and aggregating data. The second way is to merge it with its prior convolution layer from the model side. Alwani *et al.* first explore the layer fusion technique by creating a computation pyramid across adjacent layers to reduce data transfer [56]. They pipeline the computation in PE to save bandwidth, which may increase the overall latency. Additionally, extra on-chip storage is required for overlapping pyramids. These challenges are intensified when implementing such techniques in NoC-CNN. Simba [30] presents a large-scale CNN inference system using a two-level mesh NoC. They assume the weights remain stationary in PEs and only the neuron outputs are transferred across PE. Then a post-processing unit (PPU) for pooling and non-linear activation is deployed in each PE for cross-layer fusion. This can improve data locality but requires large on-chip buffers. Zhu *et al.* propose a sparse skipping technique in NoC-CNN to reduce the memory overhead for CNN inference [57]. They make the PPU an independent core in NoC and link it with off-chip memory. It can mitigate the drawback in storage scale. However, the standout PPU creates a bottleneck where every convolution output will be sent to it for pooling.

These layer fusion techniques are beneficial for reducing data movement, but they are also possible to increase end-to-end latency in NoC-CNNs. Inspired by the concept of in-network processing [53], we can offload the pooling function in PPU from PE to the network for NoC-CNN. Thereby, the pooling operation is combined with the communication in its prior layer to reduce the layer latency.

3.3 Activation in network design for NoC-DNN

3.3.1 NoC-DNN architecture overview

NoC-based DNN accelerator mainly involves three components, including memory hierarchy, network, and processing cores. **Fig.** 3.1 illustrates the NoC-DNN accelerator architecture, which is interconnected via a 2D mesh network. There are generally two types of cores. PE cores consist of the control logic, arithmetic logic units (ALUs), and local buffers, which are used for mathematical computations. Memory controller (MC) cores let NoC access the on-chip global buffer. Routers are linked with cores through the network interface (NI).

3.3.2 Non-linear activation offloading motivation

Fig. 3.2 shows a typical output stationary process for a convolutional layer in DNN, where each PE is mapped with one output neuron at a time. Therefore, the operation inside PE can be formulated as $O = F(\sum(I \times W))$. Here I represents the input features, and W represents the kernel weights. Once the MAC operation is done, the non-linear activation function F, such as ReLU or Sigmoid, is applied

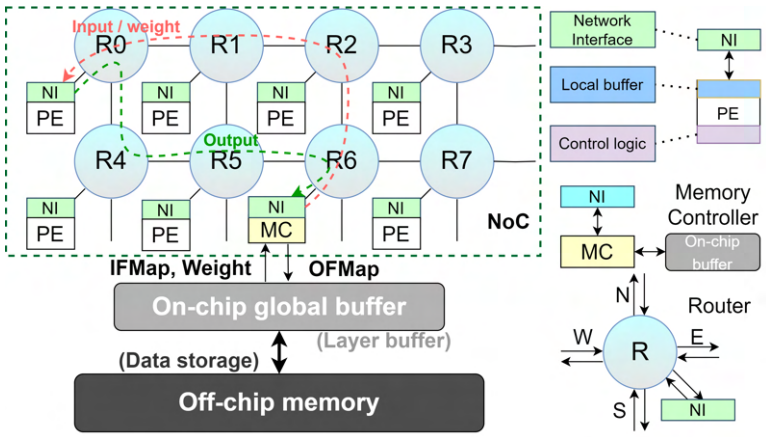


Figure 3.1: NoC-DNN accelerator architecture.

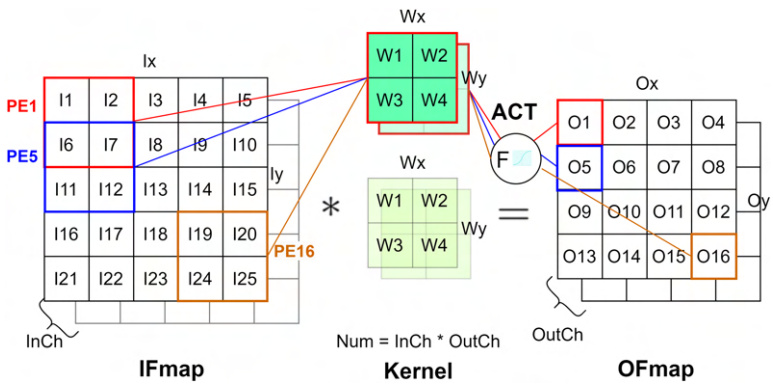


Figure 3.2: Output stationary convolution process.

to get the neuron output. Recent state-of-the-art NoC-based DNN accelerators normally place an additional non-linear activation function block inside each PE or alongside the PE array in a computation tile [28, 29, 41]. Both placements accommodate the computation process that non-linear activation happens at the end of processing one neuron output to ensure correctness.

In NoC-DNN, the inference latency consists of the processing time in PE, transmission time in NoC, and memory access time. Modern NoCs may use Quality of Service (QoS) mechanisms such as priority-based services to speed up critical packets so as to improve application performance [45, 58]. However, network congestion still exists, particularly in hot spot regions, e.g., near MC cores for gathering data, leading to higher transmission latency and likely forcing PE to stall. Inspired by the in-network processing concept, we can actually exploit such latency-induced opportunity by moving part of the computation from PE to the network. In this

case, we propose offloading the non-linear activation functions to network routers, considering that these functions are the last step of neuron computation and can be efficiently implemented using approximate computing logic or lookup tables.

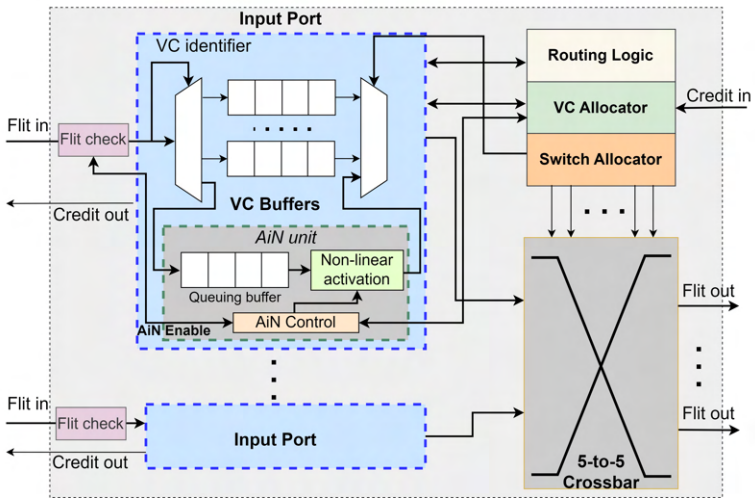


Figure 3.3: Proposed AiN VC router architecture.

3.3.3 Non-linear activation-in-network (AiN) design

Our AiN technique utilizes the packet congestion time for calculating non-linear activation. To realize our idea, we modify a canonical VC router architecture from [24]. **Fig.** 3.3 depicts the new AiN design with two additional units in the router. First is the flit check unit for input flits that controls the multiplexer for VC buffer selection. Second is the AiN unit for flit buffering and non-linear activation operations.

3.3.3.1 Flit check

In a conventional VC router, one flit follows a four-stage pipeline in the order of Routing Computation (RC), VC Allocation (VA), Switch Allocation (SA), and Crossbar Traversal (CT) [24]. AiN will only be activated if the coming flit is blocked and it requires non-linear activation. We place the flit check unit into the routing pipeline at the VA stage. This is because the neuron output packet is normally a short single-flit packet and VA happens only for the head flit.

A one-bit status tag is added to the flit header to indicate the condition for AiN. If non-linear activation is required for its payload, the tag is set to 1, otherwise, it remains 0. Besides the tag, we check the VC buffer occupancy as the normal VA stage does. Assuming that the AiN queuing buffer takes one cycle to pass the

flit, while the activation logic takes another cycle to compute, we expect at least 2 cycles of blocking in the input port as a suitable situation. Hence the flit check unit counts the existing flits in all VCs at its input port. If there are larger or equal to three flits waiting during VA stage and the queuing buffer is not full, the flit check unit grants an AiN enable signal to all flits with a tag 1. Therefore the multiplexer can pass those flits to the AiN unit instead of VC buffers.

3.3.3.2 AiN unit

The AiN unit includes a queuing buffer, a non-linear activation function block, and a controller. The queuing buffer is set as an extra VC buffer with the same size. During each SA stage, it receives a valid flit from the flit check unit if an AiN enable signal is asserted. For each cycle, if there exists a flit in the queuing buffer, the controller takes one flit out and decodes its payload according to the status tag. Then the activation function block performs non-linear activation for the payload using one cycle. After that, its tag is set to 0 and its priority in VA stage is set to high for the next arbitration.

The flit check and AiN units are placed in each of the input ports that have a neighbor router, so we use four instances if there are four directions as shown in **Fig.** 3.1. In case there is no congestion in packet trajectory, the non-linear activation will be done at the destination router. Hence an additional set of AiN units is added to the input port connected to NI in MC routers to handle the computing near memory situation.

3.3.4 Theoretical performance improvement

We discuss the theoretical improvement of the end-to-end inference latency. Since NoC-DNN uses a layer-completing workflow, we focus on the analysis for a single layer. Convolution layers and fully connected layers are common layers that require non-linear activation. To unify the analysis process, we unfold the convolution layer where each element on the output feature map represents one neuron. We assume that neuron tasks are uniformly assigned on all PEs in NoC. Theoretical inference latency improvement ratio ρ is derived by

$$\rho = 1 - \frac{T_{ar} \times r + T_c + T_{tr}}{T_{ape} + T_c + T_{tr}}, \tag{3.1}$$

where T_{ar} is the AiN processing time and r is the ratio of non-congested flits over all flits that require non-linear activation. T_{ape} represents the non-linear activation time in PE. T_c is MAC operation time in PE, and T_{tr} is the end-to-end latency for data transmission. For the same task on the NoC-DNN hardware, T_{ar}, T_{ape}, and T_c can be treated as constant. Then the upper bound of ρ appears when $r = 0$. However, $r = 0$ means AiN valid flits are all queuing in the router, which indicates a much longer T_{tr} than $r = 1$ has. In real NoC-DNN accelerator designs, one of the

essential design objectives is to optimize the communication delay. Thus we expect a larger r to shorten T_{tr} for a well-designed NoC.

3.4 Evaluation of AiN

3.4.1 Evaluation setup

To validate the proposed method, we build up a cycle-accurate behavior level NoC-DNN simulator in C++. The VC network design is adopted from the NoC simulator in [45], which is based on Garnet in Gem5 [46]. Then we create PE and MC cores and map the DNN workload as the baseline design. After that, we implement the flit check and AiN functions to VC routers following our scheme. We record the end-to-end latency in both setups for comparison. In this simulation environment, we support common DNN layers including convolutional, pooling, and fully connected layers. We also implement ReLU and Sigmoid functions in AiN units.

3.4.1.1 NoC-based DNN accelerator configuration

The network architecture consists of an 8×8 2D mesh as summarized in **Tab.** 3.1. We arrange cores in a row-major order with ID 0 to 63 and separate the whole network into four 4×4 regions. MC cores are positioned at the center of each region with IDs of (17, 18), (21, 22), (41, 42), and (45, 46). The AiN queuing buffer has a depth of 4 flits. We assume that the router operates at a higher speed than PE, with the NoC running at 2 GHz [45], and PE operates at 200 MHz as in [50].

Table 3.1: NoC-DNN configuration

Item	Amount	Configuration
NoC	64 nodes	8×8 mesh network, 2 GHz. X-Y routing. 256-bit link bandwidth. Bidirectional link.
Router	64 routers	4 VCs per port, 4-flit buffers per VC.
Core	64 cores	56 PE cores and 8 MC cores, 200 MHz.

3.4.1.2 Simulator workflow

We use a row-major mapping for DNN neurons to PEs using output stationary data flow. Each PE is assigned one output neuron in a computation round. NoC-DNN requires multiple rounds to complete one layer computation. In each round, the NoC activity to compute one neuron output can be divided into three steps:

(a) In the first step, PE sends a request packet to MC, which involves the neuron ID to fetch weights and inputs from the global buffer.

(b) In the second step, MC sends the required data packet to PE. The packet length depends on the amount of data. For a neuron with one 5×5 convolution

kernel, the payload consists of 25 inputs, 25 weights, and 1 bias. Suppose we use a 16-bit fixed-point format and the package header is 16 bits for routing information, we need $\lceil \frac{(2\times25+1)\times16+16\ (bit)}{256\ (bit/flit)} \rceil = 4$ flits.

(c) In the third step, PE sends the neuron output packet back to MC. Each PE is designed to calculate 25 MACs per PE cycle. In the baseline, the non-linear activation takes another PE cycle. In our AiN design, the activation step is offloaded to the router and it takes one router cycle.

3.4.1.3 DNN model for AiN evaluation

Table 3.2: LeNet structure for AiN in NoC-DNN

Layer	#Neuron	#Round	#OP	#ACT	ACT func.
Conv1	4704	84	$(5 \times 5) \times 4704$	4704	ReLU
MaxP1	1176	21	$(2 \times 2 \times 6) \times 1176$	-	-
Conv2	1600	29	$(5 \times 5 \times 6) \times 1600$	1600	ReLU
MaxP2	400	8	$(2 \times 2 \times 16) \times 400$	-	-
FC1	120	3	400×120	120	ReLU
FC2	84	2	120×84	84	ReLU
FC3	10	1	84×10	10	Sigmoid

We choose a well-known CNN model, LeNet-5 [44] on MNIST classification, to demonstrate our approach. LeNet consists of 7 layers in total, comprising two convolutional (Conv) layers, two pooling (MaxP) layers, and three fully connected (FC) layers. As shown in **Tab.** 3.2, we count the MAC (for Conv and FC layers) or Maximum (for Max. pooling layers) operations (OP), non-linear activation function (ACT), and computation rounds within each layer. Computation rounds are the number of output neurons divided by available PEs. In our case, the formula is $\#\text{Round} = \lceil \#\text{Neuron}/56 \rceil$. MaxP contains no non-linear activation, hence it is not relevant to our method.

3.4.2 Evaluation results

We first assess the classification results of LeNet through a comparative analysis between the outcomes from Python and our simulator. The layer output and final output consistency validate the correctness of our simulator. Then we record the end-to-end inference latency based on two setups: the baseline and our proposed approach.

We present the results in **Fig.** 3.4. The bar plot shows the end-to-end inference latency for different layers and the whole model. The line plot shows the latency improvements in percentage. Among the seven layers, two convolution layers constitute the major computational load in both tests. This observation aligns with the computational complexity of layers.

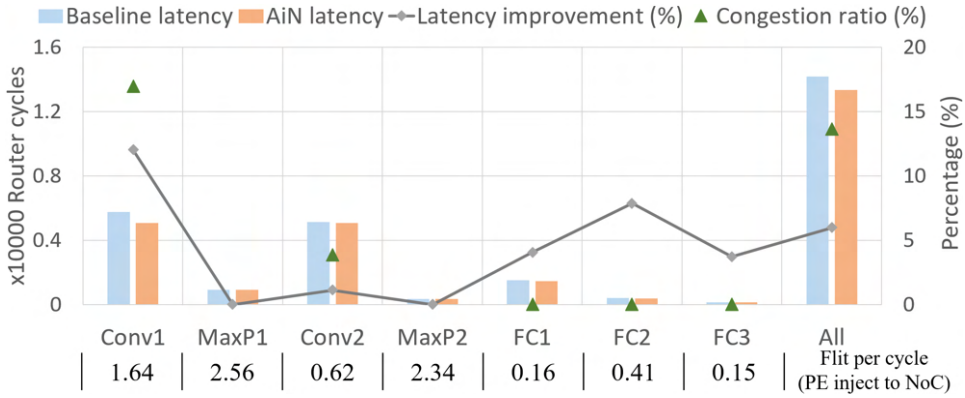


Figure 3.4: NoC-DNN evaluation results on LeNet.

Our proposed approach demonstrates a notable 5.97% latency reduction for the entire LeNet model compared with the baseline. The maximum latency improvement per layer, illustrated in **Fig.** 3.4, reaches 12.02% in the Conv1 layer. Furthermore, the FC layers exhibit improvements ranging from 3.70% to 7.84%. Since pooling layers do not involve activation computations, there is no difference between the two tests.

3.4.3 Discussion

According to **Eq.** (3.1), r and T_{tr} determine the latency improvement ratio. The averaged congestion ratio $(1-r)$ that counts the congested packets among all third-step packets from PE to MC is plotted in green in **Fig.** 3.4. Since AiN doesn't apply to pooling layers, their congestion ratio is not shown.

DNN layers with higher network activities present greater opportunities for enhancing performance by our AiN method. As shown in **Fig.** 3.4, Conv layers have more PE to MC traffic than FC layers, which increases the back pressure to block flits in NoC and results in a lower r. Thus Conv layers can benefit more from AiN than FC. Though AiN is seldom activated in FC layers with nearly zero blocking situations, they still enjoy a modest speedup. This is attributed to routers running at a higher frequency than PEs. In our case, FC layers have $r \approx 1$, and T_{ape} is ten times of T_{ar}. Comparing FC2 to the other two FCs, it has a higher flit injection rate, which means a shorter $T_c + T_{tr}$. Hence it enjoys a greater improvement ρ in consistency to **Eq.** (3.1). Conv1 has a higher congestion ratio than Conv2. Additionally, since the second-step packet size in Conv2 is six times larger than Conv1, it prolongs overall T_c and T_{tr}. This can cause PE stalls and hinder the benefits of AiN, which leads to an even smaller speedup ratio. Consequently, Conv1 enjoys better inference speedup, while Conv2 has a lower performance improvement than Conv1.

3.5 Pooling on-the-go design for NoC-CNN

A common NoC-CNN architecture [36] is depicted in **Fig.** 3.5. In the mesh network, two types of cores are connected to the network interface (NI) and router. The PE core contains the arithmetic unit and local buffers for neuron computations. The memory controller (MC) core is the portal to the on-chip buffer that transmits data to and from other PE cores through NoC. The on-chip buffer hosts the input feature map (IFMap), weight table, and output feature map (OFMap) for a layer. It updates the buffered weights after completing the current layer from off-chip memory, and OFMap is reused as the next IFMap.

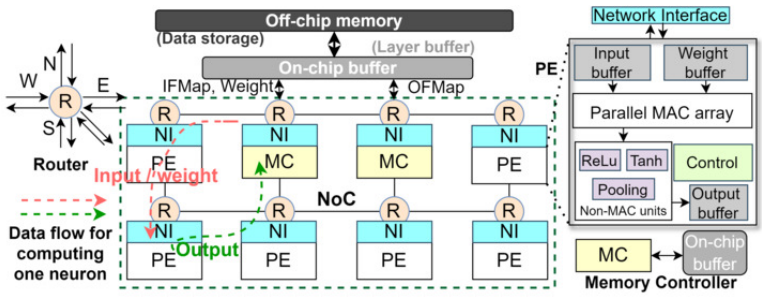


Figure 3.5: NoC-based CNN accelerator schematic.

To accommodate various layer types, we utilize the unfolding technique in an output-oriented manner similar to the assumption in [36]. Each layer is represented by a batch of output neurons that equals the size of OFMap. NoC-CNN maps these neurons to PEs during CNN model loading. Assuming a layer's OFMap size is M, and NoC has k PEs, with one PE computing one neuron per round, it necessitates $R = \lceil M/k \rceil$ rounds to complete this layer.

3.5.1 Pooling on-the-go approach

In the pooling standalone NoC-CNN design, the pooling operations are processed in PE. For example, in a DNNoC-like [36] workflow, PE will save the convolution layer (Conv.) outputs to off-chip memory and load it again for pooling operations. This costs extra data moving overhead and lowers the utilization of PE. Therefore, we propose a pooling on-the-go approach to combine the pooling layer computation during transmission, minimizing redundant data movements. **Fig.** 3.6 illustrated the standalone and on-the-go pooling processes, both starting at T_0. For standalone pooling, the Conv. OFMap is completed at T_1. Then it serves as the pooling layer IFMap and finishes at T_2. Alternatively, in our approach, the pooling operation is done during the transmission of Conv. OFMap from PE to MC. Then MC receives the pooling OFMap at T_2'. The latency improvement of the combined layer is $T_2 - T_2'$.

To realize pooling on-the-go in NoC-CNN, there are three possible positions other than PE, including router, NI, and MC. In flit-based virtual channel (VC) routers, data as payload are unknown to existing routing logic. Therefore, extra decoding units are required for each port in case the pooling happens in the router, introducing a large area and latency overhead. NI and MC are equivalent to pooling block implementation as they are near the end of the data path. Since MC is directly connected to the global buffer, it runs in a slower clock domain than NI. Hence NI is a more suitable position, and only NI connected to MC needs modification. **Fig.** 3.6 also compares the data path of two pooling designs, where the pooling IFMap transfer is removed in our pooling on-the-go approach.

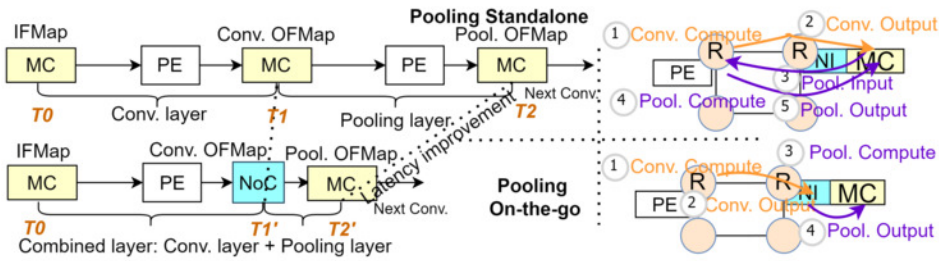


Figure 3.6: Timeline (left) and data path (right) of standalone and on-the-go pooling.

3.5.2 Pooling on-the-go implementation

A canonical NI consists of two symmetric ports as the interface between core and router [24]. The original *outport* design is shown at the top of **Fig.** 3.7, where core messages are turned into flits and injected into the router. We develop a pooling on-the-go block to offload the pooling layer operations from PE and integrate it in our new *inport* design. The pooling block structure is depicted in the lower half of **Fig.** 3.7. We have a temporary buffer to store intermediate pooling values, a pooling function block for calculation, and a controller to manage the whole process. Two multiplexers are used to enable or bypass the new pooling block.

Message Header A 1-bit **tag** is added to the message header to judge whether a message requires the new pooling service and control the multiplexer. Note that only a combination of a pooling layer after a convolution layer can be optimized from our approach. To correctly perform pooling operations, the message header also includes a 1-bit signal that tells the pooling function, e.g. max pooling and average pooling. Then the channel and ID are given to identify the location in the pooling layer OFMap instead of the initial Conv. OFMap. The remaining bits are reserved for different pooling kernel sizes.

Local Pooling Table This is a key component to store the temporary results for pooling, which is organized as a buffer table that can access an entire row at a time.

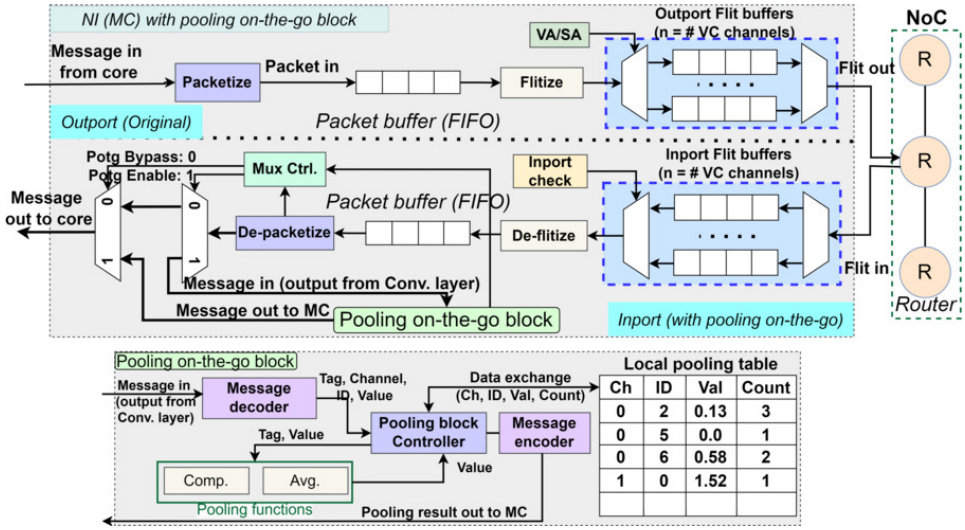


Figure 3.7: Network interface with proposed pooling block and the block design.

In each row, there are four elements, representing channel, ID, value, and count. "Count" indicates the number of messages received for the same channel and ID. We don't have to allocate the full size of pooling OFMap in the pooling table at once. Since there are multiple MCs in NoC, each will communicate with a fixed number of PEs. Then in each computation round, the number of messages that NI receives is also fixed. So the pooling table only needs the space to hold these messages in one round simultaneously. For example, if in the NoC-CNN, five PEs are communicating with this NI to reach MC, then we only need five rows in the pooling table.

Pooling Block Controller **Fig.** 3.8 shows the workflow of the pooling on-the-go controller. The input message from the packet buffer in *inport* is sent to the pooling block when the valid **tag** is given. Then it performs the following steps:

1. The decoder extracts function, channel, ID, and value from the message.
2. Controller checks the pooling table for index comparison on channel and ID.
3. If no matching entry, data is written to an empty row with Count set to 1.
4. If a matching entry exists, we calculate the pooling result of the input value and the stored value. The Count of this row increases by one.
5. If the Count is smaller than the pooling kernel size, the result is written back. Otherwise, a new message is created to deliver the final pooling result.

The controller iteratively processes the valid messages until the whole pooling layer OFMap is completed and stored in the on-chip buffer. Our design ensures

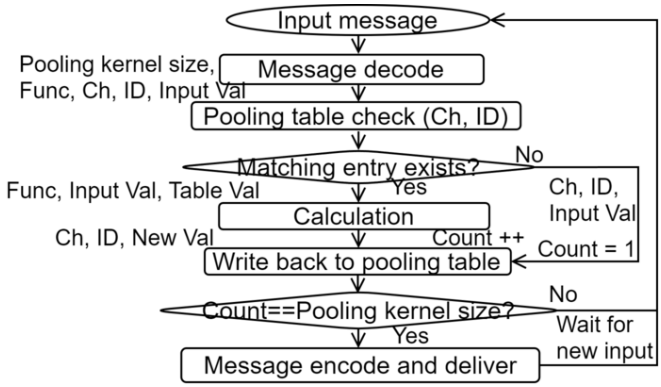


Figure 3.8: Pooling block controller workflow.

that the pooling operation is finished in NI during the transmission of its prior convolution outputs.

3.6 Evaluation of pooling on-the-go approach

3.6.1 Experimental Setup

We build a cycle-accurate NoC-CNN simulator based on the NoC simulator in [45] to evaluate our approach. The VC NoC architecture is adopted from Gem5-Garnet [46]. We implement the pooling standalone baseline. Then we deploy our pooling on-the-go functions in the simulator for tests. The inference result and layer latency when running the same CNN model on baseline NoC-CNN and our approach are recorded to validate the correctness and evaluate the latency improvement. Based on **Fig.** 3.6, we define the latency reduction P of the pooling layer and the speedup ρ of the combined layer using our approach as

$$P = \frac{T_2 - T_2'}{T_2 - T_1} \times 100\%, \quad \rho = \frac{T_2}{T_2'}. \tag{3.2}$$

Simulation System Configuration The default NoC-CNN configurations are summarized in **Tab.** 3.3. We arrange cores in a row-major order, for example, cores with ID 0 to 7 are in the first row. Then we separate the whole network into four identical 4×4 subareas. MCs are positioned at each subarea's center with ID 17, 18, 21, 22, 41, 42, 45, and 46. We simulate an ASIC NoC-CNN in which the NoC components (router, NI) can run at 2 GHz [30, 45]. PE and MC are usually slower than routers since they have more complex combinational circuits for calculation. The default PE frequency is 200 MHz as the UNPU in [50]. We also change the PE frequency to 500 MHz and 1 GHz to simulate different hardware designs [30]. The latency results are in the unit of NoC cycles.

Table 3.3: NoC-CNN default configuration

Item	Amount	Configuration
NoC	64 nodes	8×8 mesh network. 2 GHz. X-Y routing. 256-bit link bandwidth. Bidirectional link.
Router	64 routers	4 VCs per port, 4-flit buffers per VC.
Core	64 cores	56 PEs and 8 MCs. 25 MACs per PE cycle. 200 MHz.

We select two representative CNN models, the small LeNet model [44] and the large VGG16 model [59], for evaluation. They both contain the combination of a pooling layer after a convolution layer that our approach applies. The input size is $32 \times 32 \times 1$ for LeNet and $224 \times 224 \times 3$ for VGG16.

LeNet comprises two combined layers that our approach applies. **Tab.** 3.4a gives the parameters for these layers: L1 (Conv1+MaxP1), L2 (Conv2+MaxP2). ϕ shows the proportion for computation (#OP) of pooling over the total combined layer. VGG16 has 13 convolution layers, five of which are followed by a max pooling layer that can use the pooling on-the-go method. **Tab.** 3.4b summarizes the parameters and the proportion of pooling computation of these five layers. For ease of simulation, we sample VGG16 with the same kernels but smaller feature maps, then scale them up to get the latency for the full model.

Table 3.4: Layer parameters for pooling on-the-go

(a) LeNet

Layer		Neurons	Kernel size	#OP	ϕ (%)
L1	Conv1	4704	5×5	(1×5×5)×4704	3.846
	MaxP1	1176	2×2	(2×2)×1176	
L2	Conv2	1600	5×5	(6×5×5)×1600	0.662
	MaxP2	400	2×2	(2×2)×400	

(b) VGG16

Layer		Neurons	Kernel size	#OP	ϕ (%)
L1	Conv2	3211264	3×3	(64×3×3)×3211264	0.173
	MaxP1	802816	2×2	(2×2)×802816	
L2	Conv4	1605632	3×3	(128×3×3)×1605632	0.087
	MaxP2	401408	2×2	(2×2)×401408	
L3	Conv7	802816	3×3	(256×3×3)×802816	0.043
	MaxP3	200704	2×2	(2×2)×200704	
L4	Conv10	401408	3×3	(512×3×3)×401408	0.022
	MaxP4	100352	2×2	(2×2)×100352	
L5	Conv13	100352	3×3	(512×3×3)×100352	0.022
	MaxP5	25088	2×2	(2×2)×25088	

3.6.2 Experimental Results

The LeNet latency on baseline NoC-CNN and pooling on-the-go approach (with label P) are plotted in **Fig.** 3.9. In three evaluated situations, the average latency reductions for MaxP1 and MaxP2 are 98.98% and 97.39%. Speedup ρ for the combined layers L1 and L2 are 1.148x and 1.054x. The more significant improvement in L1 is primarily due to the higher pooling proportion ϕ compared to L2. The core frequency determines the computation latency. Thus a shorter T_1 and T_2 in **Eq.** (3.2) are expected for faster PE, while $(T_2' - T_1)$ remains the same. Consequently, the best speedup appears for the 1 GHz setup, which is 1.09x for the full model, 1.16x for L1, and 1.06x for L2. Our observation validates that the proposed approach can achieve a better speedup when PE frequency is higher.

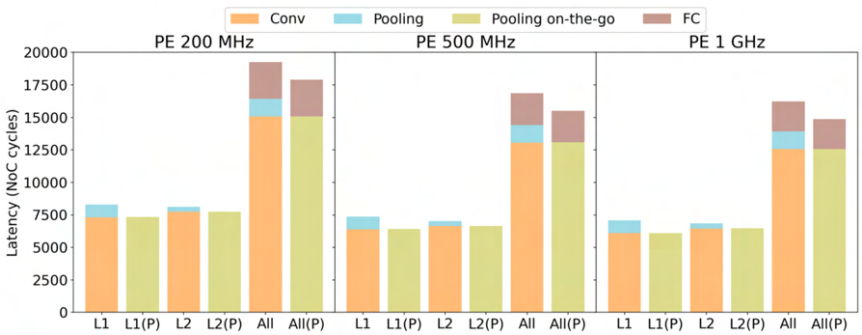


Figure 3.9: Inference latency of LeNet.

Fig. 3.10 presents the inference latency of L1 to L5 in VGG16. The full VGG16 model needs 392 million NoC cycles on the baseline NoC-CNN, so we don't plot it here. For five pooling layers MaxP1 to MaxP5, the latency reductions are all over 99.9%. The average speedup of the combined layers from L1 to L5 ranges from 1.016x to 1.0015x. The overall speedup on VGG16 is 1.0036x, 1.0035x, and 1.003x, respectively, when the PE frequency is 1 GHz, 500 MHz, and 200 MHz. We can see that the pooling layer latency is almost eliminated using our approach as we expected. However, the pooling operation only occupies a very small proportion of VGG16. For example in the 1 GHz PE setup, it takes around 1.2 million cycles, which is 0.359% of the full model. Therefore, the speedup of our approach on VGG16 is not obvious.

3.6.3 Discussion

From the evaluation results, the pooling layer latency is almost eliminated when our approach is applied. The overall speedup depends on the workload proportion of the pooling layer in the CNN model. In LeNet, this proportion is way higher than in VGG16. Thus our method can achieve much better speedup results in

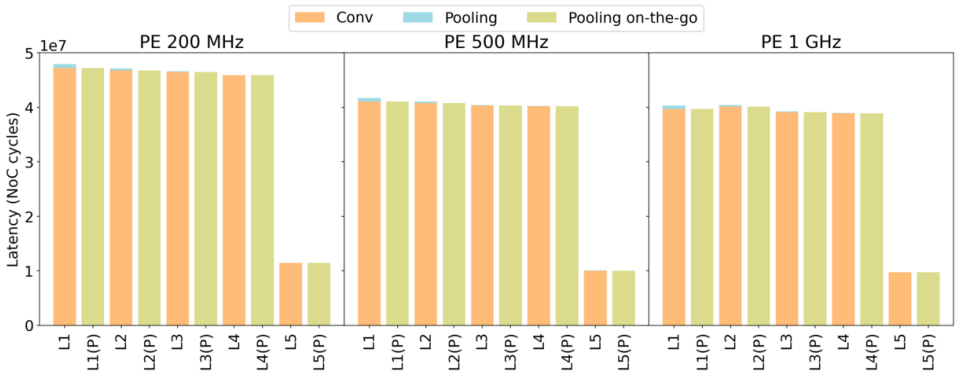


Figure 3.10: Inference latency of VGG16.

LeNet compared with VGG16. Moreover, our approach replaces the standalone pooling units in PE with pooling on-the-go blocks in NI connected to MC. MC constitutes a small fraction of total cores in NoC-CNN, and the area for the extra pooling table and simple controller in the block is not significant. So the proposed approach can maintain or even save area for NoC-CNN. Also, the reduced memory accesses for the standalone pooling layer can improve energy efficiency. Regarding the communication overhead, since the Conv. output packet only contains one output neuron value, the added message header will not affect the packet size as it remains a single-flit packet.

Therefore, the proposed pooling on-the-go approach can improve the performance and efficiency of NoC-CNNs, especially when the target CNNs have lightweight convolution layers followed by normal pooling layers.

3.7 Summay

In this chapter, two in-network processing inspired NoC-DNN architectural designs are presented. Both of them aims to improve the inference performance in terms of end-to-end latency for DNN workloads.

A modified VC router micro-architecture is designed to realize the non-linear activation offloading approach for DNN inference latency optimization. The observed latency improvement aligns with the theoretical equation, particularly benefiting DNN layers with frequent NoC accesses and shorter processing times, such as small-kernel convolution, through the proposed AiN. The AiN design reveals the opportunity for in-network processing in DNN accelerator designs. We believe that other operations can also be offloaded to the network to reduce inference latency.

We also proposed a pooling on-the-go approach with a unique pooling block design in the network interface of NoC-CNN. This method aims to do pooling operations during the transmission of its prior layer outputs, thereby minimizing the

pooling layer processing latency. Compared with common standalone pooling, our approach reduces superfluous data transfers to optimize efficiency. The simulation results show that our approach can almost eliminate the pooling layer latency. This opens the opportunity for cross-layer optimization using in-network processing in NoC-CNN designs.

Their RTL implementation and tradeoffs between timing and power/area are worth investigating in future work.

Chapter 4

Machine Learning Applications for Embedded Sensor Systems

Besides the hardware accelerator designs for efficient DNN inference, the exploration of machine learning applications in embedded systems has gained significant traction. Embedded systems, characterized by their resource-constrained environments and real-time operational requirements, present unique challenges and opportunities for deploying machine learning techniques. These systems are increasingly integral to a wide range of applications, spanning from smart devices to wearable health monitoring solutions.

In this context, we introduce two practical machine learning applications for embedded sensor systems, demonstrating their potential to address real-world challenges effectively. Specifically, we focus on the domain of lower-limb prostheses, a critical area for enhancing mobility and quality of life for individuals with limb loss. The primary objective is to enable real-time collection and processing of dynamic pressure data within the prosthetic socket. By leveraging machine learning methods on such embedded sensor systems, we can analyze intra-socket pressure distributions to improve comfort and provide quantitative evidence for guiding the design of prosthetic sockets, aligning their performance with the amputees' gait dynamics.

Through these applications, we highlight the capability of machine learning in embedded sensor systems that can transform sensor data into actionable insights. This paves the way for smart decision systems that cater to the specific needs of lower-limb prosthetics.

4.1 Background

There are approximately 40 million amputees worldwide and the number is expected to increase due to an aging population and a relatively higher incidence of

This chapter is based on the material from papers 4 to 7.

diabetes and vascular diseases [60]. Transfemoral patients constitute a substantial part of all limb amputations [61]. The demand for effective prosthetic socket design grows rapidly due to an increasing number of amputees and the lack of efficiency in designing comfortable sockets. This is essential to improve rehabilitation comfort and safety for lower-limb amputees.

In response to this grand challenge, a current trend is to enable a quantitative data-analytic approach to assist prosthetists in evaluating patient comfort, and rectifying socket shape [61]. With such an approach, wearable intra-socket pressure measurement systems have been interesting to facilitate socket fit. One example is shown in **Fig.** 4.1, where the electronic measurement system is attached to the prosthetic socket to collect data. These data can then be analyzed to generate relevant information to help prosthetists to monitor, evaluate the performance of the socket and adjust the socket design for comfort and comfort enhancement.

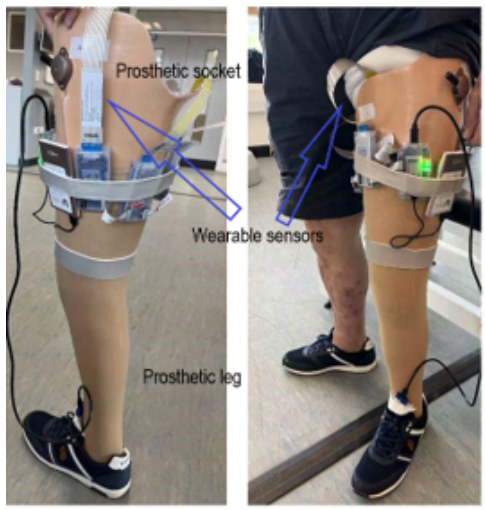


Figure 4.1: Wearable sensors embedded in socket for transfemoral amputee.

4.1.1 Wearable intra-socket pressure measurement system

Fig. 4.2 depicts an overview of our pressure-sensing system, which consists of three subsystems as follows:

1. Sensor subsystem: Conformable printed pressure sensors using QTSS™ (Quantum Technology Super Sensors) materials [62] were developed as sensor strips, each strip hosting multiple sensors. This allows us to flexibly deploy sensor strips in desired locations to capture interesting points of measurement across the inner socket.

2. Electronic measurement subsystem: The purpose of measurement is to acquire the generated data from the sensors. It mainly consists of an Analog/Digital Conversion (ADC) interface and an embedded ESP32-based processing platform.

3. Storage and visualization subsystem: The acquired data are stored locally in an SD card and transmitted via the Internet to a remote database. The visualization front end works together with the data analysis toolboxes to show preliminary data analysis results.

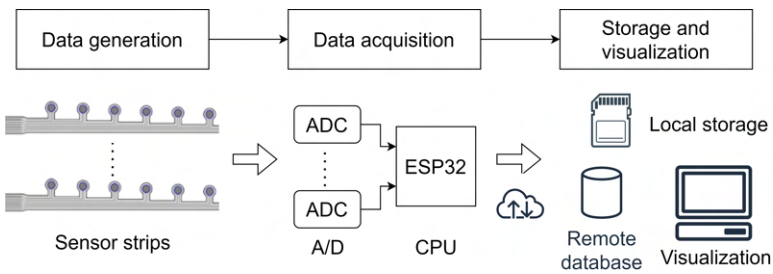


Figure 4.2: The pressure-sensing system.

4.1.1.1 Sensor subsystem

In our system, we use QTSS™ sensors, which are flexible wearable sensors. The materials are magnetite-based, anisotropic and change from 'insulator' to 'conductor' under pressure, becoming increasingly conductive in response to the amount of applied force. Thus a QTSS™ sensor works as a variable resistor whose resistance R changes with the applied pressure P. Using this property, we can conveniently measure the voltage V on the variable resistor in our electronic measurement system, thus obtaining V as a function g of pressure P, $V = g(P)$. In practical use, we first characterize the function g for each sensor. Then, given a measured voltage V in an actual clinical trial, we can derive its corresponding pressure P by the inverse function $P = g^{-1}(V)$.

We adopt the strip-based sensor organization. Compared to an array-based sensor organization, it gives more flexibility to place the sensor strips with different density in a socket. **Fig.** 4.3 shows a sensor strip with eight QTSS™ sensors printed on a polyethylene terephthalate substrate.

One challenge with printed sensors is that their pressure-response characteristic can drift. Calibration is thus needed for each sensor. In our approach, we use an off-site major calibration in the test room, combined with on-site minor calibration using a hand-held pressing device after the sensors are mounted in the socket. The major calibration is to obtain the sensor pressure characteristic curve g, and the on-

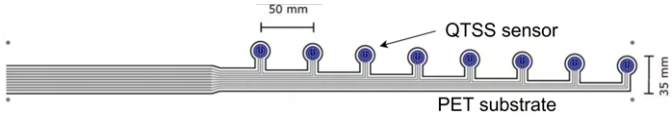


Figure 4.3: Sensor strip

site minor calibration is to tune g using proportional shift according to the sensor's present property.

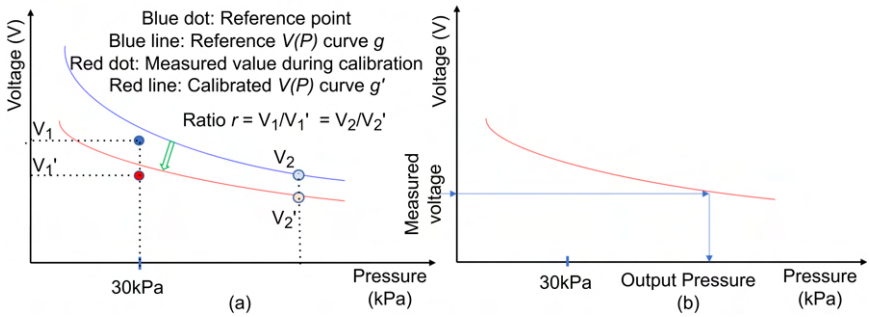


Figure 4.4: Principle of sensor calibration

Fig. 4.4(a) shows the principle of sensor calibration. The sensor pressure characteristic function g was obtained through the major calibration, which is used as the reference curve. Because of differences in the measuring conditions, the measurement point generated by the hand-held pressing device is typically deviated slightly from the position of the major calibration curve. During the on-site minor calibration, the hand-held pressing device was used to apply the known pressure of 30 kPa on a sensor to get the 'present' voltage (V_1' vs. V_1) under this pressure. Thus we obtain a shifting ratio $r = V_1'/V_1$. The entire curve of g is then proportionally shifted towards the new measurement point, indicated by the red curve g'. The shifting ratio for all points on the curve is equal to $r = V_1'/V_1$. As shown in **Fig.** 4.4(b), the calibrated curve g' is used to derive the pressure given a measured voltage.

4.1.1.2 Electronic measurement subsystem

Fig. 4.5 gives a schematic view of the electronic measurement system, which consists of two main parts: ADC interface and ESP-32 microprocessor. The pressure sensor functions as a variable resistor whose value changes in response to the force exerted on it. Due to its serial connection to another resistor with a constant value, the circuit works as a voltage divider. When force applied on the sensor varies, the voltage divider generates varying analog voltage, which is then converted into digital levels through an ADC channel. Afterwards, the digitalized level signals

are sampled from the SPI (Serial Peripheral Interface) bus by ESP32 under clock synchronization.

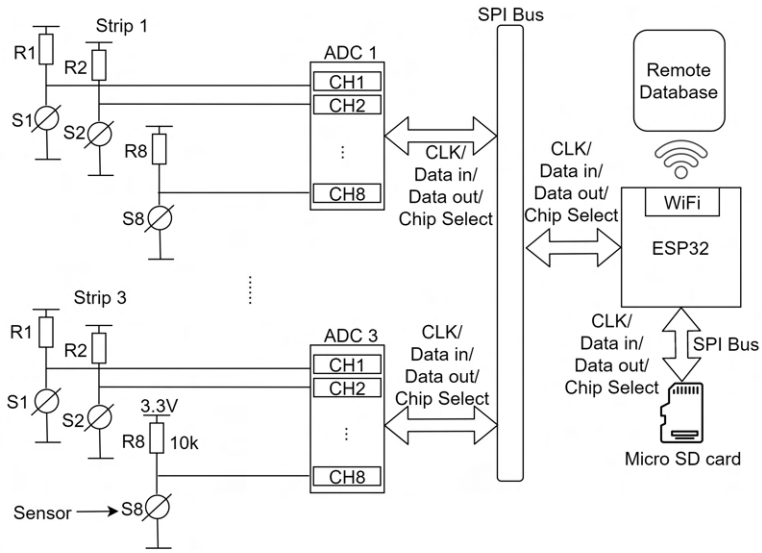


Figure 4.5: Dual-core parallel processing under the producer-consumer paradigm

The ADC is a key component. It shall be accurate enough so that we can get data with sufficient precision. Additionally, we need to consider its safe operation under unexpected circumstances, such as over voltage supply, and interfacing option with the microprocessor. With the above considerations, instead of using built-in ADC on the embedded SoC, we use an external ADC module, MCP3208, which is an 8-channel 12-bit ADC module. The 12-bit resolution gives a sufficiently high resolution to reflect meaningful pressure variation. The ADC module also satisfies the need for SPI interfacing, flexibility in voltage division, and over-voltage protection for each channel.

Numerous options exist for developing a microprocessor-based embedded system, including platforms like Arduino and Raspberry Pi. After evaluating performance, cost, and size, the ESP-WROOM32E from Espressif Systems was selected. This SoC integrates a dual-core Xtensa 240 MHz 32-bit microprocessor, which is well-suited for data acquisition and transmission tasks. One core manages data readout from ADC channels, while the other handles data storage and real-time transmission to a remote server. Communication with peripheral devices is facilitated through the SPI bus, enabling easy integration of additional sensor strips or peripherals, limited only by the SoC's I/O pins. The system also features an SD card interface for local data storage and Wi-Fi connectivity for remote database storage. To minimize interference, the SD card operates on a dedicated SPI bus.

Rather than bare-metal program, the data processing software runs on top of

the freeRTOS [63]. It is organized into three tasks: data acquisition, data buffer management, data communication. The dual-core ESP32 SoC enables efficient task distribution, with tasks divided across the cores in a pipeline structure. A producer-consumer communication paradigm is employed, where the data acquisition task on CPU1 acts as the producer, and the data transmission task on CPU0 serves as the consumer. The two tasks communicate via a shared data queue, implemented using freeRTOS xQueue functions. In our current software loop, sampling one round of data (one frame or packet) from all sensing peripherals may take 50 ms, 20 ms, 10 ms (the OS task granularity is 10 ms) by configuration. Thus the data frame sampling rate can be 20 Hz, 50 Hz and 100 Hz, respectively.

4.1.1.3 Data storage and visualization subsystem

The data communication and storage task transmits sampled data to the InfluxDB database on the remote server and to the SD card as local backup. This redundant scheme allows the measurement data to be accessible through the database in real time and from the SD card after data collection. This ensures that even under network communication disturbances from Wi-Fi or Internet, the SD card can serve as a safe backup.

To enhance usability, a graphical user interface (GUI) has been developed for interactive visualization of the time-series data. The GUI supports various display formats, allowing users to view data segmented by sensor strip, individual sensors, or specific activities performed by each amputee.

4.1.2 Clustering algorithms on embedded sensor platform

Among various analytical tasks, clustering has proven effective in uncovering features within time series sensor data. While solving this problem on the embedded platform is almost absent because of the limitation of computational resources on the edge. Given the resource constraints of embedded microprocessors, these methods rely on pre-trained models developed on more powerful systems, with inference performed locally at the edge.

To find out the feasibility of such time series clustering methods on embedded sensor platforms, we need to evaluate their performance, taking both accuracy and data inference time under consideration. We focus on two different clustering methods, the representative partitioning approach of K-means algorithm and the neural network approach of Self-Organizing Map (SOM) algorithm. Besides, both the Euclidean distance and the Dynamic Time Warping (DTW) algorithm are employed as the similarity measurement.

This approach aligns seamlessly with our wearable intra-socket pressure measurement system, as it can benefit from efficient on-device data analysis to deliver timely insights and enhance operational reliability. One application that employs clustering-based method for sensor deployment is later introduced in Section 4.2.

4.1.2.1 Background on selected clustering algorithms

The main idea of K-means algorithm [64] is to minimize the total distance between all objects inside cluster while maximizing the distance between clusters. The principle solution of K-means is based on the Expectation-Maximization algorithm. Each iteration consists of two key steps: in the expectation step, data points are assigned to clusters based on their proximity to the current centroids; in the maximization step, the centroids are updated to reflect the mean positions of the newly assigned clusters. This iterative process continues until the clusters converge to an optimal configuration.

SOM [65] is a neural network model designed for clustering tasks, emphasizing the geometric relationships within a dataset. SOM consists of two layers: an input layer and an output (competitive) layer. The input layer acts as the perception channel, processing the original time series, while the output layer represents the desired clusters through its neurons. Training SOM involves a competitive process where each input is assigned to the best matching unit (BMU), the neuron most similar to the input. The weights of the BMU are then updated, along with those of its neighboring neurons, based on their proximity to the BMU. This weight adjustment can be performed using stochastic or batch training methods, refining the clusters iteratively.

K-means algorithm has advantages of fast convergence and straightforward computation and interpretation of clustering results. However, its performance is sensitive to the choice of the number of clusters (K) and the initialization of centroids, making it prone to falling into local minima. In contrast, the model-based SOM doesn't require every neuron in the output layer to represent a cluster, allowing for more flexibility. Its cooperative training process helps mitigate the influence of noisy data, but this comes at the potential cost of reduced clustering accuracy in some cases.

DTW [66] is an elastic distance measure designed for comparing discrete sequences, making it well-suited for analyzing time-series data collected from embedded sensor systems. DTW excels in similarity search tasks by performing one-to-many comparisons, allowing it to adapt to the stretching or compression of time-series sequences. In the embedded sensor platform, sensor data could be misaligned due to variations in sensor positions or reading order. The DTW algorithm effectively addresses this issue by eliminating distortions, enabling more accurate comparisons and analyses of the collected data.

4.1.2.2 Evaluation process

Fig. 4.6 summarizes the implementation and experiment process. For a dedicated dataset, we would do normalization before feeding it into the clustering models. Then models are trained on a computer to obtain the relative optimal ones. These trained models are then implemented on the embedded processor for the time evaluation of the inference process. The distance comparison in K-means and SOM

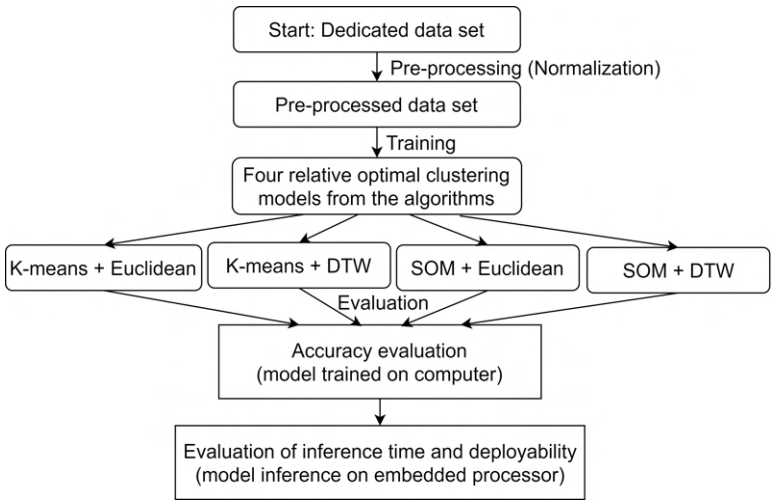


Figure 4.6: Evaluation process overview.

algorithms can be replaced by DTW from the original Euclidean distance. Hence in total, four combinations of the algorithms, K-means with Euclidean metric, K-means with DTW, SOM with Euclidean metric and SOM with DTW, are under evaluation as shown in the diagram.

The experiment models are evaluated on a single core of ESP32, which is also the microprocessor in our pressure measurement system. If we want to insert the clustering function into this system, it should be scheduled in the time gap of the recurrent transmission tasks on the same core of the ESP32, meanwhile, the sensor reading process will occupy the other core. In our sensor platform setup, it collects pressure data from 40 sensors at a rate of 50 Hz. A packet of 40 frames, each frame with a length of 50 data points is generated in one second. Under the lab situation, the data transmission task costs an average of 77.1 ms to handle the packet round trip between the server and the sensor system, while a peak of 163.8 ms is recorded. Consider that the data are already in the memory when we apply the clustering algorithm to them before the packet is sent, we have an upper bound of around $1000 - 163.8 = 836.2$ ms for the inference process. This setup provides a time baseline.

4.1.2.3 Evaluation results

To evaluate the clustering algorithms, we use well-organized and widely used public data sets. Several entries are picked up from the UCR time series classification archive [67]. They are labeled and formatted with a short description of the data source. For this evaluation, we focused on the BME dataset (3 classes, 180 samples and 128 points per sample) and the three Cricket datasets (12 classes, 780 samples

and 300 points per sample), all categorized under motion data. This selection aligns with one of the essential targets of our embedded sensor platform, which is to analyze pressure data related to motions. The software timer is set up to record the calculation part of the inference process on the ESP32, and the whole dataset is shuffled for the evaluation. The averaged time represents the inference time for one sample.

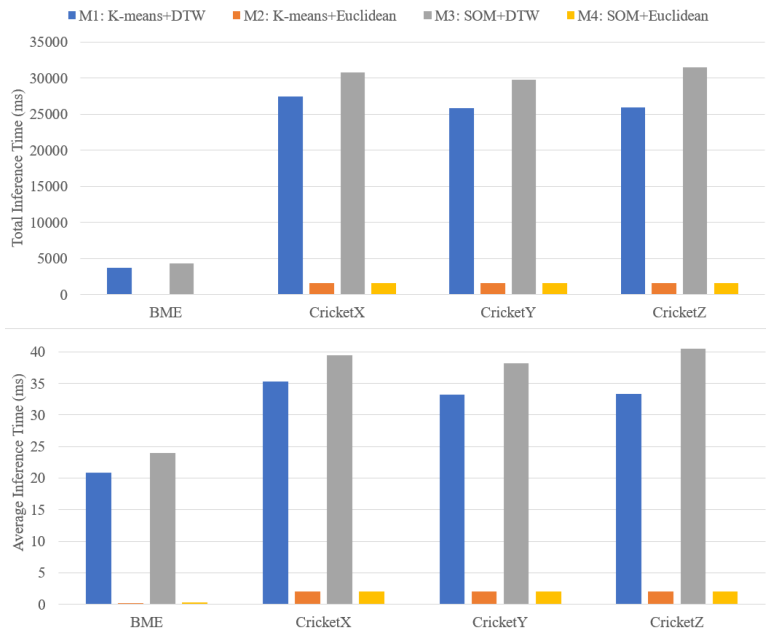


Figure 4.7: Total and average inference time on ESP32.

The results of the experiments on ESP32 are shown in **Fig.** 4.7, in which model M1 is the K-means with DTW, model M2 is the K-means with Euclidean distance, model M3 is the SOM with DTW, and the model M4 is the SOM with Euclidean distance. From this figure, we can find that the time used in SOM is 12% to 40% longer than the K-means. Generally, the time used with DTW is 19% to 93% longer than with Euclidean distance. A sample in BME contains 128 data points, while a sample in Cricket dataset contains 300 samples. Obviously that the sample with a shorter length will consume less time in the inference phase for the same clustering model. The longest total inference time here is 31.5s when using the model M3 on the CricketZ dataset. Generally, we can make a cluster model using the SOM with DTW on a sample of 128 points for three classes in 24 ms, or a time series sample of 300 points for twelve classes in 40 ms. Compared with the fastest M2 among four models, the inference time for a sample of 128 points for three classes in 0.22 ms, or a sample of 300 points for twelve classes in 2 ms.

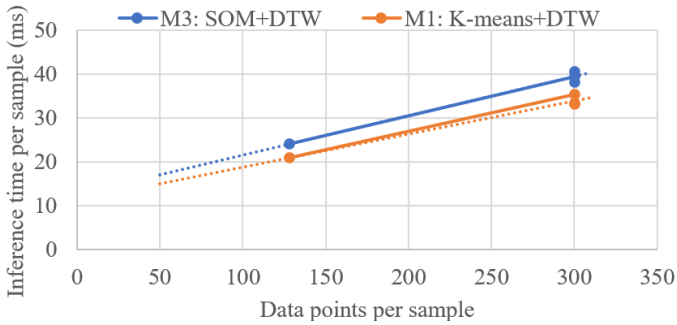


Figure 4.8: Relationship between sample length and inference time.

Here we discuss the time requirement for deploying these time series clustering models in our embedded sensor platform study case. Using interpolation of the evaluation results for the two DTW enabled models, M1 and M3, with the longest inference time results among four models, the prediction lines are shown in **Fig.** 4.8. Then we can estimate that the frame with length of 50 data points takes at most 17 ms to complete, which is 680 (17×40) ms for a whole 40-frame packet. Hence for all the implemented models, the inference time in our study case should be less than 680 ms. By comparing with the given deadline of 836.2 ms, the inference task is quite acceptable to be scheduled in our sensor platform. The data size is around 16 KB for one packet, with the expected weight matrix of 1.2 KB for a three-cluster model in double type, which can be completely loaded into the 4 MB memory of ESP32. Overall, it is possible to deploy all four time series clustering models on our target sensor platform.

4.2 Application 1: Redundancy reduction for sensor deployment

Uneven and high interfacial pressure distribution of the prosthetic socket can lead to many discomforts in amputees' daily lives [68], while the development of the sensing technology [69] helps identify such stresses. In Section 4.1.1, we introduced the commercial sensing systems and our SocketSense in-house solution for measuring intra-socket pressure. Such sensor systems usually have high density to ensure the coverage of the whole stump. The high-density coverage may, however, introduce redundancy, which increases cost and unnecessary complexity. Therefore, detecting and removing the dispensable sensors while maintaining the effectiveness of the pressure measurement becomes crucial in prosthetic sockets' pressure studies.

This application intends to reduce the local redundancy in the sensor deployment by proposing a clustering-based analysis method that can locate the unnecessary sensors by evaluating the data from a predefined sensor layout and then provide suggestions to lower the current sensor density. The clustering method, such as a

Self-Organizing Map (SOM) [65], is employed to find the clustering results of the sensor data. The most common one is selected as the redundancy detection model for the sensory system. Then, the similarity metrics, such as the Pearson correlation coefficient (PCC) [70], are applied to indicate the unessential sensors in the relevant regions and evaluate the efficacy of the modification on the sensor layout. Jenson–Shannon Divergence (JSD) [71] is applied as a validation metric from the pressure distribution perspective to check if the new sensor layout is credible.

Overall, the proposed method aims to improve the sensor deployment no matter what kind of pressure sensor or which individual amputee is being treated. Our redundancy reduction method has the potential to be employed in smart socket design, for example, in a closed control loop of an actuation system for efficient data acquisition with reduced weight and complexity of the final system.

4.2.1 Related work

The measurement of the interfacial pressure is critical to understanding the comfort level of patients in the prosthetic socket design. Several researchers have performed physical testing on the socket in the prosthetic study. Commuri *et al.* [72] have assessed the pressure distribution during walking inside the transfemoral socket with triaxial force sensors mounted on the socket wall. Tran *et al.* [73] analyzed the pressure and the shear stresses of an amputee with the sensors embedded in the socket and visualized the pressure map according to the sensor readings.

The clinical trial from Ali *et al.* investigated the interfacial pressure during walking with hundreds of resistive transducers placed on the residual limbs [74], and they also studied the pressure during the stair ascent and descent activities using the same sensory system [75]. Although these researchers used a large number of sensors during the test, their analysis normally focused on the mean peak pressure, the standard deviation and other statistical features of the pressure in four major areas, such as anterior, medial, posterior and lateral. The results in the sub-regions under these major areas are quite similar, which indicates the redundancy in the sensor layout. This implies that some of the sensors can be removed inside these regions, and the important features of pressure distribution are not lost. When reducing the sensor density, the time and cost of the measurement can also be decreased.

There have been attempts to reduce the sensor density when the researchers are building their own sensing system. Jasni *et al.* [76] developed the in-socket sensory system based on the analysis of Tekscan F-Socket sensors. Their method was to locate the sensors in the most active muscle areas determined by a muscle assessment of a particular amputee. However, they did not investigate the redundancy that existed in sensor placement inside the socket as their research was not focused on solving the sensor redundancy problem.

The fuzzy clustering method has been used for redundancy detection in the environment monitoring aspect, where the air pollution data were collected monthly from all over the world [77]. Their analysis method is not suitable for intra-socket

measurements since the sensors inside the socket have much higher density and frequency. The premise of clustering, however, can reveal the similarity in the sensor readings and thus detect the redundancy in sensor layout. In our case, SOM is a good choice as it can preserve the topology information on the 2D grid [65]. Compared with the K-means algorithm [64] and fuzzy c-means clustering algorithm [78], SOM allows empty clusters in the output clusters. To select the essential sensors inside the socket according to the clusters, SOM provides more stable clustering results for redundancy detection.

Specifically, our method uses the SOM clustering algorithm to detect redundancy, a similarity-based method to guide the removal of non-essential sensors. We use a distribution similarity metric to validate that the redundancy removal results in acceptable information loss.

4.2.2 The wearable sensor deployment problem in prosthetic sockets

The wearable sensors for interfacial pressure measurement are made flexible and rather thin so they can be easily mounted in prosthetic sockets. **Fig.** 4.9 gives an example of the wearable sensor system placed in a transfemoral socket, where the sensors are fixed by adhesive tapes. Usually, the researchers use hundreds of sensors to reach high coverage on the surface of the whole stump. However, handling those sensors in the clinical test for one subject can take a few hours with many types of measurement equipment attached to provide enough channels for sensor reading. These tests may continue with the fitting of the socket shape for several iterations, which is a time-consuming activity [79].

Hence, the sensor deployment appears to be a challenging problem as a trade-off between the measuring coverage and the efficiency. The efficiency is improved by using a smaller number of sensors and less time in clinical trials with an acceptable loss in interfacial pressure measurement.

Since large redundancy exists in the pressure data analysis process, the pressure distribution is quite uniform in some of the areas with less muscle contraction [76]. This also implies redundancy in the data acquisition process. How to detect and remove the redundancy of the sensor placement during the data collection then becomes a key point to this problem. A redundancy reduction method is required to deal with it and to reduce the sensor density by removing the unessential sensors.

After the removal of the sensors inside the socket, it is necessary to validate that similar pressure features such as the distribution and mean pressure can still be obtained with the optimized sensor deployment.

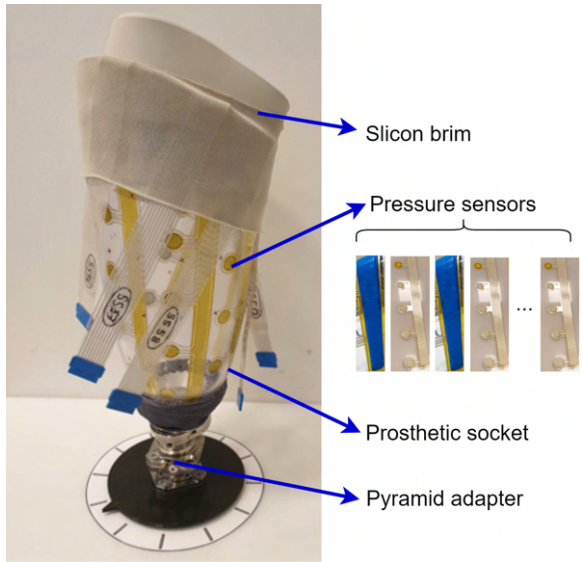


Figure 4.9: Prosthetic check socket with sensors attached.

4.2.3 Sensor redundancy reduction method

4.2.3.1 Overview

To optimize the sensor deployment for the socket interfacial pressure measurement, a key factor is to reduce the redundancy in the sensor layout. Since there has been extraordinary variability in the amputee situation and prosthetic socket configuration, there are no common sensor deploying rules. We proposed a clustering-based method to detect and remove the sensor redundancy for interfacial pressure measurement for amputees, and it could be general guidance for socket sensor deployment.

Based on the practical experience of prosthetists, a preliminary sensor layout is established for the general cases. Then, an intra-socket measurement with this layout on a real patient is required to obtain the initial pressure readings. According to the pressure data, we aim to find the similarity between them since the redundant sensors would generate correlated pressure curves when they are adjacent to each other. Intuitively the unsupervised clustering methods can be employed for this task, which separates the sensors into groups according to their similarities and indicate the existence of unnecessary sensors in the local region.

Fig. 4.10 shows the overall process of the clustering-based analysis method for reducing redundant sensors, which consists of four functional steps to generate a new sensor deployment scheme.

1. **Data input:** The sensor data may have various formats depending on the

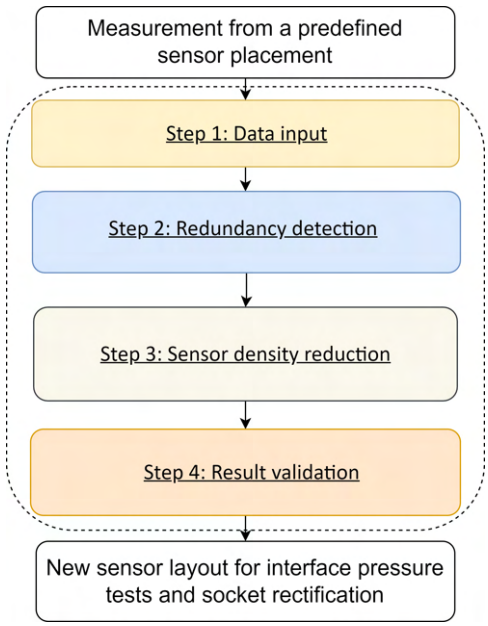


Figure 4.10: The clustering-based analysis method process.

sensory system, and they need to be converted to the actual pressure in the same unit. After that, the pressure data are cleaned and sliced into frames, which contain several gait cycles, to maintain enough information during the dynamic tests. These frames constitute the initial dataset for the analysis method.

2. **Redundancy detection:** By adjusting the parameters of the SOM and feeding with different data frames from the initial dataset, multiple clustering results are learned. Among them, one common clustering result (which appears the most for all the data frames and model configurations) is picked up as the target model to detect the sensor redundancy for the input case.

3. **Sensor density reduction:** According to the redundancy detection model, the local redundancy in current placement is recognized. Considering the actual requirements and the capability of the sensory system, the unnecessary sensors can be removed from the corresponding clusters. The similarity metrics such as PCC can be used to guide the selection.

4. **Result validation:** The sensor removal results need to be evaluated based on the choice from step 3. The pressure distribution over the whole test from the initial sensor layout will be compared with readings from the reserved sensors, using entropy-based metrics such as the Jenson–Shannon Divergence

(JSD). After the posterior evaluation, we can determine how dependable our sensor selection is.

The last step of the proposed method would provide a new sensor layout for the input case with a pruned layout. If the number of sensors to be maintained changes because of the test conditions and the patient situation known from the prosthetist, we return to step 3 and go through step 4 again to obtain another sensor removal suggestion.

Fig. 4.11 provides an example of the redundancy removal process of a dedicated sensor deployment with our method. The original layout is a long strip with ten sensors on it, which are labeled with numbers 1 to 10. By applying the clustering method, we can separate them into two clusters, sensors 1 to 4 are grouped in the blue cluster, and sensors 5 to 10 are grouped in the orange one. Suppose the prosthetist would like to keep two sensors for the blue cluster and four for the orange cluster based on the redundancy detection model, then the similarity metric is checked within the two clusters. Finally, sensors 2, 3 from the blue cluster and sensors 6, 7, 9, 10 from the orange cluster are reserved to form the new layout. Finally, validation metrics such as JSD and mean pressure are used to compare the information loss between the cluster and the selected sensors. The new sensor placement can be applied for future tests.

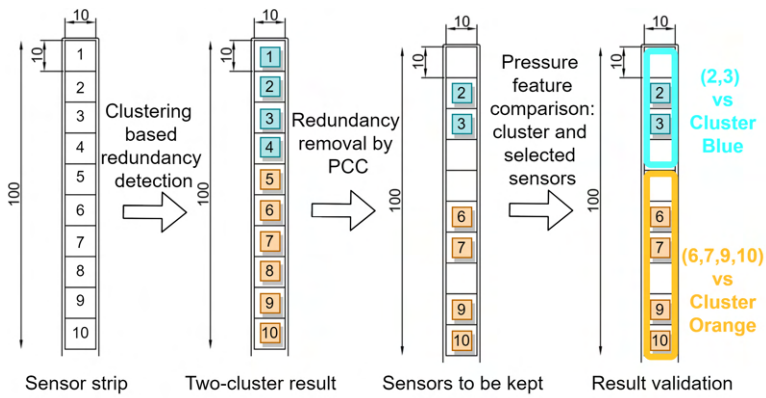


Figure 4.11: An example of a redundancy removal procedure on a sensor strip consisting of 10 sensors.

4.2.3.2 Redundancy detection and clustering algorithms

The second step in our method uses the clustering algorithm to build the sensor redundancy detection model. Here, we will introduce the Self-Organizing Maps (SOM) [65] as the main method for generating the clustering results.

SOM is a two-layer neural network model that contains the input layer and competitive layer. The number of neurons in the competitive layer represents the number of clusters for output. Suppose we have an input set of time series x_i, with the length of n, and the weights connect the two layers between the ith input neuron and the jth output neuron $w_{ij}, i = 1, \cdots, n, j = 1, \cdots, k$. k is the number of expected clusters. τ is the total iteration number. Then, the learning process can be described as follows:

1. **Initialization:** The weights of the SOM are first initialized, e.g., with some small random numbers.

2. **Competition:** Each input will find its best matching unit using the similarity metric. The winning unit is called the best matching unit (BMU).

3. **Cooperation:** The BMU decides the range of its neighbors to update weights. Suppose we use the Gaussian distribution, $\sigma(t) = \sigma_0 \exp(-\frac{t}{\tau})$ is the parameter decay as iteration grows, with the initial standard deviation σ_0 and current iteration number t. Then, the update distribution T of node j is given by $T_{j,BMU}(t) = \exp(-\frac{D^2}{2\sigma(t)^2})$, and D is the geometry distance between node j and BMU. The closer neighbor will obtain a larger update.

4. **Adaptation:** The weights of the neurons are updated by $\Delta w_{ij} = \eta(t) \times T_{j,BMU}(t) \times (x_i - w_{ij})$. In the equation, the learning rate is defined as $\eta(t) = \eta_0 \exp(-\frac{t}{\tau})$.

5. **Iteration:** Go back to the above steps from the competition until τ iterations are complete. The final winning neurons of the input time series are their clusters.

SOM uses a kind of global decision-making strategy, which could avoid falling into the local minimum. It can still be stable and accurate when the noise data exists. On the other hand, it relies on the learning parameters and can generate different clustering results on the same dataset. Hence, we have to train the SOM on multiple data frames sliced from the original dataset and treat the most common clustering result as the final redundancy detection model on all frames.

In addition, we use other clustering methods such as the classic K-means algorithm to perform the cross-validation. The K-means algorithm aims to minimize the distance between all input vectors to the corresponding centroids based on the least-squares method [64]. It will be used on the same data frames to see if there is a common one among all the frames. We compare it with the SOM result to evaluate the reliability of the redundancy detection model.

4.2.3.3 Metrics for guiding sensor removal

The Pearson correlation coefficient (PCC) is one of the most popular similarity metrics used by experimental researchers [80]. The equation of PCC [81] between

two series is calculated as (4.1). The X_i and Y_i are the ith data points on time series X and Y. $\bar{X}$ and $\bar{Y}$ are the average values of X and Y.

$$\mathrm{PCC}(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{4.1}$$

PCC is a measure of linear correlation between X and Y and it ranges from -1 to 1. A high PCC close to 1 shows a linear relationship between X and Y with a positive slope, which implies a similar trend of X and Y.

Based on PCC, we can suggest the removal of unessential sensors. Suppose we have a sensor dataset and the related redundancy detection model. **Fig.** 4.12 shows the sensor removal procedure. For a cluster containing m sensors, the mean values of all sensor data in that cluster are calculated as *centroid_all_sensors*. If we select k sensors to be kept, there are C_m^k combinations. k should be a value between 1 and m, and it is flexibly set based on the knowledge of prosthetists. The mean value of the ith combination of sensors to be kept in that cluster is computed as *centroid_ith_selection*. We then calculate the PCC for each pair of *centroid_all_sensors* and *centroid_ith_selection*. The selection with higher PCC is better. Assuming the jth combination has the highest PCC, we use it as the final choice.

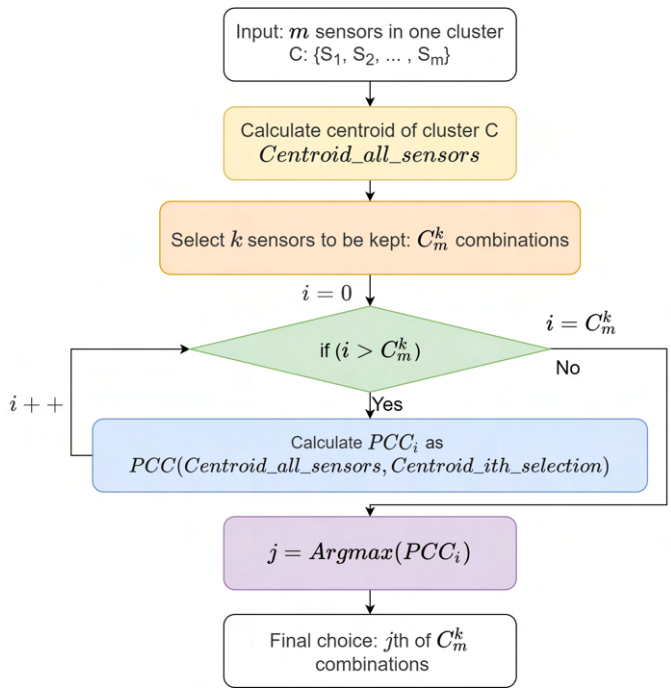


Figure 4.12: PCC-based sensor removal procedure.

For one specific number k, our proposed method selects k sensors to be kept and remove other sensors. Our proposed approach provides flexibility for prosthetists to select the number of sensors to be kept, as the method possesses the scalability to find the k sensors to be kept without a limitation on the value of k.

4.2.3.4 Validation after sensor removal

To validate whether the pressure readings after the removal of redundant sensors can still represent those from the initial sensor layout, we need to perform a posterior check based on the selected sensors. As the pressure distribution among gait cycles is an important factor for socket fitting [69], it can be used as a reference for validation. For each cluster we obtain from the redundancy detection model, a pressure distribution P can be obtained on the whole input dataset from all sensors in this cluster. On the other hand, for the selected sensors in that cluster, they will constitute another pressure distribution Q over the same walking phase. Then, the difference between these two distributions can be found by the statistical divergence.

In our case, since the P contains the information of Q and they are in the same cluster, they should have enough overlapping, then the Jenson–Shannon Divergence (JSD) [71] can be a suitable similarity metric between P and Q. JSD is a bounded symmetry metric based on the Kullback–Leibler Divergence (KL) [82], which means $JSD(P||Q) = JSD(Q||P)$ with a value between 0 and 1. When $JSD(P||Q)$ close to 0, P is similar to Q. The calculation of JSD is given in (4.2), in which $p(x)$ and $q(x)$ are probabilities from distribution P and Q.

$$\begin{aligned} KL(P||Q) &= \sum p(x) \log \frac{p(x)}{q(x)}, \\ M &= \frac{1}{2}(P+Q), \\ JSD(P||Q) &= \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M). \end{aligned} \tag{4.2}$$

By examining the possible combinations of the selected sensors, we can have multiple pressure distribution Qs. Then, we calculate the corresponding JSDs and evaluate if the selection has good coverage of the pressure distribution compared with the original sensor layout. The selections with lower JSD can present the information from all the sensors better, and we use this metric as a validation of our approach. In addition to the pressure distribution, the mean pressure during walking is also an important feature for the prosthetist to investigate [75]. This is also included as a validation metric.

4.2.4 Case study on sensor removal

4.2.4.1 Sensor data acquisition

To show the efficacy of the presented analysis method, we conduct a case study. A set of pressure data is acquired from a clinical trial on a transfemoral amputation patient. The trial was conducted after the protocols had been reviewed and approved by responsible national authority [1].

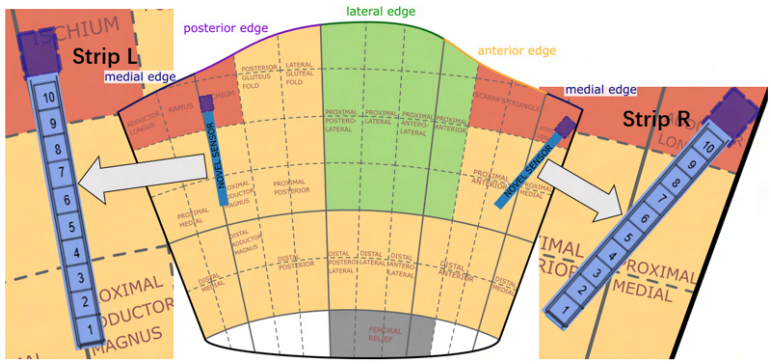


Figure 4.13: A 2D representation of a prosthetic socket with Novel [83] sensor placement (2 strips with 10 sensors on each strip) inside the socket.

The socket used in the experiment is manufactured for the user according to a scanned positive mold of the subject's residual limb. The main material of the socket is transparent Polyethylene Terephthalate Glycol (PETG plastic). It is a replica of the patient's own direct socket [84] manufactured by Össur with a soft silicone brim. **Fig.** 4.13 draws the 2D coordinate map of the socket according to the shape of the socket. The research from Neumann *et al.* [85] obtained a dynamic socket pressure mapping in a suitable socket, which indicates the windows for pressure sensitive areas. The regions divided by different colors in Figure 4.13 refer to their hypotheses, where red indicates high pressure and green indicates low pressure on average. The blue strip indicates the sensor position along the posterior edge (strip L) and the medial edge (strip R). The measurement device to read pressure data from sensors is Pliance [86] from Novel Electronics Inc., and two sensor strips used in the test are Novel S2006 [83], with 10 capacitive pressure sensors on each strip (Single sensor area: $10 \times 10mm^2$, Sensor pressure range: $2 - 200KPa$, Thickness $< 1.2mm$, Pliance reading range setting: $0 - 64KPa$, Pliance reading frequency setting: 100Hz).

[1] The clinical investigation was conducted in compliance with Icelandic regulations and guidelines and in accordance with the ethical principles that have their origin in the Declaration of Helsinki, and the protocol, CII2020121754, was approved by the Ethics Committee IRB VSN 19-083.

The conducted test is the indoor level ground walking at the patient's self-selected speed. The first 100 steps are for the sensor calibration, which are excluded from the test results. Then the pressure is recorded at a rate of 100 Hz during the rest of the test. To build the dataset, we take 30 seconds of the pressure readings from both of the positions, namely dataset L for the posterior region and R for the medial region. Each of the datasets contains ten sensors and with a length of 3000 points.

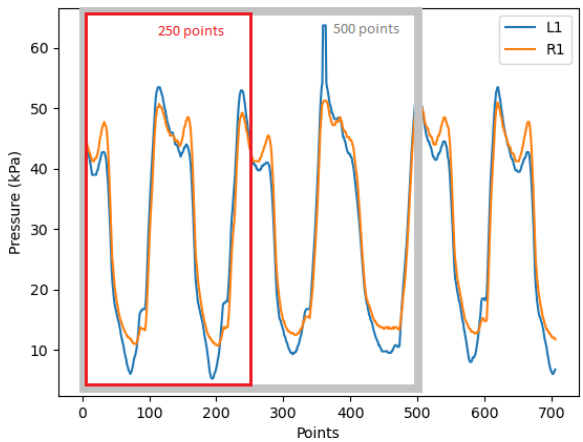


Figure 4.14: A sensor data frame in length of 250 and 500 points. ($L1$ is the first sensor in dataset L, $R1$ is the first sensor in dataset R)

After that, we slice the dataset into multiple frames of shorter length. The frame length has an influence on redundancy detection as it contains a different number of gaits. In **Fig.** 4.14, the trough to trough is equivalent to one full gait cycle. For example, 250 data points comprise two gait cycles and 500 data points contain around four gait cycles, which is framed by the red and grey rectangle respectively. Other researchers like Jasni *et al.* [76] have used a frame size of five seconds for in-socket pressure tests on transfemoral amputees. In our experiment, we also choose 500 data points (5 seconds) as the frame length as it includes four to five gait cycles depending on the patient's speed, which can provide enough information on the pressure changes during walking. We will also test the frame length of 250 points and compare it with the redundancy detection result of 500 points.

4.2.4.2 Redundancy detection

The clustering methods are applied on the six frames, each with length of 500 points, from the dataset L and R respectively to build the redundancy detection models. For the SOM, we set the output layer as a 3×3 rectangular grid, as the number of input entries is ten. The learning rate is set to 0.2 and the SOM runs

for 10 iterations. Then six clustering results would be generated for each of the datasets, and the most common one will be selected as the final clustering result.

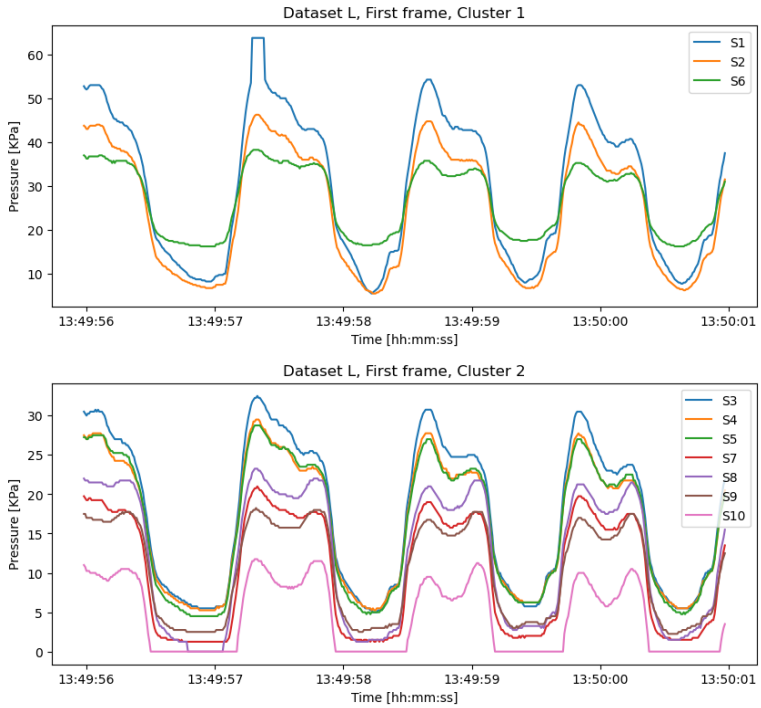


Figure 4.15: Cluster1 and Cluster2 of the first data frame from dataset L in posterior region.

Fig. 4.15 plots the clustering result for the first frame on dataset L, which has two non-empty clusters. Sensor number 1, 2, and 6 are in the same cluster, while the other seven sensors are in the second cluster. Actually, for six frames of L, we only get one different clustering result on the fifth frame, which is $\{1, 2, 3, 6\}$ and $\{4, 5, 7, 8, 9, 10\}$. Other five frames are clustered by $C_{l_a} : \{1, 2, 6\}$ and $C_{l_b} : \{3, 4, 5, 7, 8, 9, 10\}$. So we consider this two-cluster result as the redundancy detection model $M_L(C_{l_a}, C_{l_b})$ for L.

Similarly, the SOM algorithm gives the clustering results $C_{r_a} : \{1, 2, 4, 5, 6\}$ and $C_{r_b} : \{3, 7, 8, 9, 10\}$ for all six data frames in the medial region dataset R. The consistency of the clustering result proves that the way to divide sensors in such groups is reasonable. Then we can determine the redundancy detection model $M_R(C_{r_a}, C_{r_b})$. Finally, we have M_L and M_R for the sensor removal steps on dataset L and R, respectively.

Considering the same dataset with 12 frames, each with length of 250 points. With the same parameters for the SOM, the most common clustering result of

L is $C_{l_a} : \{1, 2, 6\}$, $C_{l_b} : \{3, 4, 5\}$ and $C_{l_c} : \{7, 8, 9, 10\}$, in which 7 out of 12 frames are the same. The most common clustering result of R is $C_{r_a} : \{1, 2, 4, 5, 6\}$ and $C_{r_b} : \{3, 7, 8, 9, 10\}$, in which 4 out of 12 frames are the same, while some of the frames even have four non-empty clusters after clustering. Although the final clustering results of this test are quite close to the M_L and M_R we get from the 500-point frame experiment, we can see that the clustering results are quite unstable among different frames from the same strip. So the smaller frame size can lead to unstable clustering results, and 500 points per frame is a good choice to obtain a relatively consistent result over the whole dataset.

In addition, we use the K-means on all the six frames in length of 500 points on the same dataset L and R. Since we have the two-cluster model from SOM, here we set K to 2 and run the algorithm for 10 iterations as well. The SOM model on dataset L, $M_L : C_{l_a} : \{1, 2, 6\}$ and $C_{l_b} : \{3, 4, 5, 7, 8, 9, 10\}$, appears 3 times out of the 6 frames in K-means clustering. The SOM model on dataset R, $M_R : C_{r_a} : \{1, 2, 4, 5, 6\}$ and $C_{r_b} : \{3, 7, 8, 9, 10\}$, also appears 3 times out of 6 in K-means clustering. Though the K-means model only has half the number of frames in both datasets with the common clustering results, they contain the same clusters as the SOM results. Therefore the redundancy detection model on these two datasets can be determined as M_L and M_R respectively.

4.2.4.3 Sensor removal results

Our method aims to provide suggestion for redundancy removal with k sensors left, in which prosthetists can define k by themselves. We showcase the results of choosing $k = 1$ and $k = 2$.

(1) Keep only one sensor in a cluster: To find the best single sensor of each cluster we calculated PCC when comparing with the curve from selected sensors and the average curve from all sensors in one cluster.

Table 4.1: Results for selecting one sensor in dataset L

Sensor	Cluster	PCC
a_0	C_{l_a}	0.9986
a_1	C_{l_a}	0.9997
a_2	C_{l_a}	0.9919
b_0	C_{l_b}	0.9925
b_1	C_{l_b}	0.9950
b_2	C_{l_b}	0.9977
b_3	C_{l_b}	0.9979
b_4	C_{l_b}	0.9936
b_5	C_{l_b}	0.9922
b_6	C_{l_b}	0.9597

For dataset L, there are three sensor selections in C_{l_a} and seven sensor choices

in C_{l_b}. We compute ten PCCs for each selection and repeat the calculation six times for six data frames. **Tab.** 4.1 gives the average PCC for all six data frames, in which a_0 to a_2 are from C_{l_a}, and b_0 to b_6 are from C_{l_b}. The red value means the best score of the metric among all choices.

The PCCs in cluster C_{l_a} are all higher then 0.99, which shows the similarity to the centroid of this cluster. We choose the sensor a_1 with highest PCC which is 0.9997 to be the sensor kept. For cluster C_{l_b}, sensor b_3 with PCC of 0.9979 is the winner in PCC scores, so we choose b_3 to be kept. The same result evaluation process is done on the sensor strip R. For five sensors in C_{r_a} and the other five sensors in C_{r_b}, a_2 and b_2 are selected after evaluating the PCC.

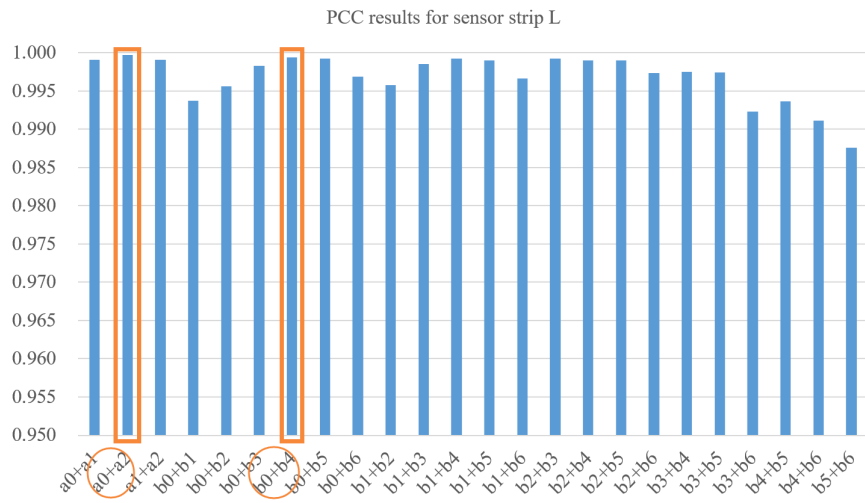


Figure 4.16: Results for selecting two sensors in dataset L.

(2) Keep two sensors in a cluster: If the sensor designer chooses to leave two sensors for each cluster, we apply the same strategies to find the best two sensors. For C_{l_a}, there are 3 sensor selections and C_{l_a} contains 21 sensor pairs. **Fig.** 4.16 shows the average results on six frames for all these 24 possible pairs. The best results for each cluster are marked. For C_{l_a} we choose the combination of $a_0 + a_2$, and for C_{l_b} the pair of $b_0 + b_4$ is kept at last. Similarly, we can choose $a_0 + a_3$ for C_{r_a} and $b_1 + b_3$ for C_{r_b} on the sensor strip R.

The high PCC provides evidence that the data has high similarity. That means the pressure curves generated by selected sensors have high potential to replace the original layout of sensors. The result of keeping one or two sensors are given as an example, while this approach could also suggest combinations with more sensors based on the opinion of the prosthetists.

4.2.4.4 Validation

To validate the redundancy removal result, we concatenate the data frames back into the whole dataset with 3000 points. Then the pressure distribution of the whole test can be presented by the histogram of the probability density. Since the sensors have a reading range from 0 to 64 KPa, we set the number of bins to 64, and then calculate the $JSD(P||Q)$. P is the pressure distribution of the corresponding cluster, which represents the initial sensor layout. Q is the pressure distribution of the selected sensors from the redundancy removal step.

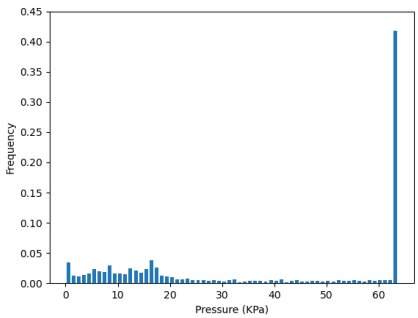


Figure 4.17: Pressure distribution from one-sensor Rb_2 in C_{r_b}.

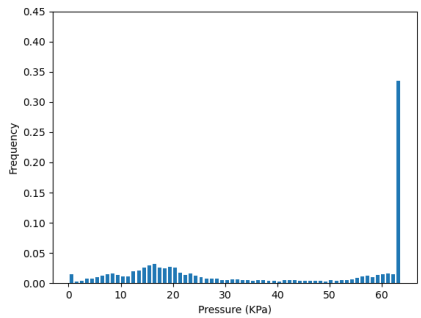


Figure 4.18: Pressure distribution from all sensors in C_{r_b}.

Fig. 4.17 plots the pressure distribution from the one-sensor selection of the cluster C_{r_b} on strip R, which is Rb_2. **Fig.** 4.18 plots the pressure distribution of all sensors in cluster C_{r_b}. We can see they both have a high distribution on the saturation pressure of the sensor, which is 64 kPa.

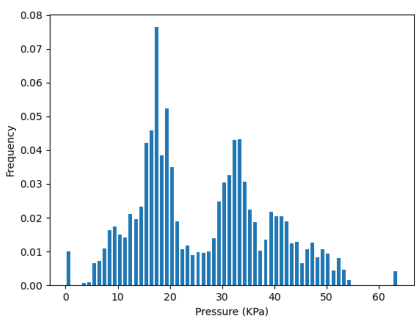


Figure 4.19: Pressure distribution from two-sensor La_0, La_2 in C_{l_a}.

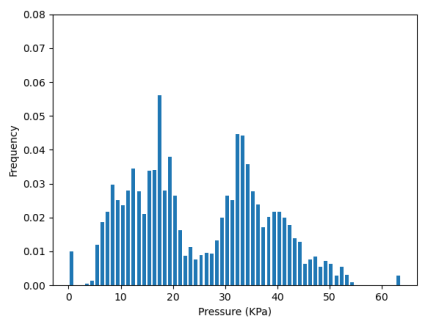


Figure 4.20: Pressure distribution from all sensors in C_{l_a}.

Similarly, **Fig.** 4.19 plots the pressure distribution from the two-sensor selection of the cluster C_{l_a} on strip L, which contains La_0 and La_2. It has a very similar distribution shape with the pressure distribution plot of all sensors in C_{l_a} during our test, as shown in **Fig.** 4.20.

Tab. 4.2 shows the JSD validation results of the selected sensor combinations after the redundancy removal procedure. **Tab.** 4.3 shows the validation results of the sensor selections with the minimum JSD values among all possible choices inside the corresponding clusters. The differences between these two tables are labeled in red. For the one-sensor selection, we can see that two of them are different from the suggested choice from the redundancy removal section, which are in C_{l_b} and C_{r_a}. For the two-sensor selection, we find one different selection in C_{r_a}. The two differences that occur in cluster C_{r_a} have an insignificant gap in JSD (< 0.06), and the validation results are also relatively low (< 0.1). That means the suggested sensor selections from the redundancy removal procedure are still credible in these two cases.

However, we observe a larger difference in the one-sensor selection in C_{l_b}, which indicates to only selecting one sensor from the cluster C_{l_b} is not an optimized solution. Since the two-sensor result is consistent in this cluster, it is better to keep two sensors rather than only selecting one.

Table 4.2: JSD from the sensor removal results.

	C_{l_a}	C_{l_b}	C_{r_a}	C_{r_b}
1-sensor selection	La_1	Lb_3	Ra_2	Rb_2
JSD	0.084	0.224	0.092	0.045
2-sensor selection	La_0, La_2	Lb_0, Lb_4	Ra_0, Ra_3	Rb_1, Rb_4
JSD	0.018	0.036	0.017	0.010

Table 4.3: The sensor selection with the smallest JSD among all choices.

	C_{l_a}	C_{l_b}	C_{r_a}	C_{r_b}
1-sensor selection	La_1	Lb_4	Ra_3	Rb_2
JSD	0.084	0.134	0.039	0.045
2-sensor selection	La_0, La_2	Lb_0, Lb_4	Ra_1, Ra_2	Rb_1, Rb_4
JSD	0.018	0.036	0.009	0.010

The mean pressure during walking is also useful for prosthetists [85] when analyzing the sensor data. Here we will compare the average pressure of the four clusters from the redundancy detection model and the sensors after removal on the whole walking dataset. **Fig.** 4.21 plots the mean pressure of the initial sensor deployment in each of the redundancy detection clusters and the selected sensor combinations from the sensor removal section. From this figure, we can see that the mean pressure of the two-sensor selection is very close to the original readings, with an average variation of 5% in all four clusters. C_{l_b} has the highest variation

of about 18.9%. For the one-sensor selection, the average variation is around 10%, and C_{l_b} still has the highest difference of 16.7%. That means the prosthetist may require more sensors in area C_{l_b} to get a more accurate mean pressure reading. In general, the two-sensor selection is better than the one-sensor selection for mean pressure measurement, which also meets our expectation.

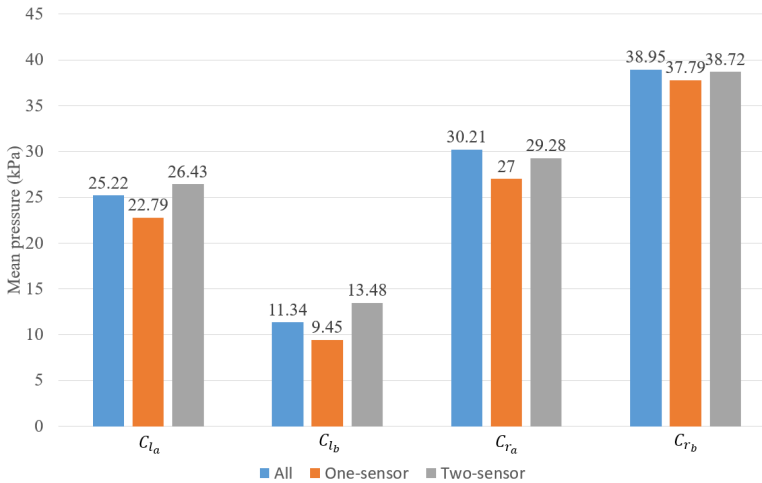


Figure 4.21: Validation of the mean pressure.

Overall, our clustering-based analysis method can guide the removal of unessential sensors in the sensory system with high-density elements. With high PCC, the information of trends is proved to be kept. The JSD shows the high similarity of pressure distribution between the sensors to be kept and the original sensors, and the mean pressure validation shows satisfying results after sensor removal. These indicate that our clustering-based method to remove redundancy is effective.

4.3 Application 2: Amputee gait phase recognition

One of the main tasks of lower-limb prosthesis systems is to ensure the tight coupling between the human limb and the artificial prosthetic socket, especially under dynamic situations [87, 88]. Biomechanical studies have demonstrated significant differences in the human lower limb joint kinematic and kinetic across various locomotion patterns and gait phases [89]. Consequently, the efficacy of prosthesis systems hinges on their ability to perceive leg movements accurately, thereby imposing stringent demands on gait analysis.

The gait analysis based on intra-socket dynamic pressure measurement is a popular approach to assist prosthetists in identifying various impairments during walking [90, 91] and achieve a better fit of the socket [60]. Gait activities can be treated as discrete gait phases in a cycle. Accurate gait phase recognition is a

key component of successful gait analysis. Other applications such as the active prosthetic exoskeletons [92–94] and auto-adjustable socket [95,96] also need precise gait phase recognition for their control logic.

In recent years, various approaches for gait phase detection have been developed. Most of them are based on wearable sensor measurements including inertial measurement units (IMU), foot switches, electromyography sensors (EMG), and their combinations [97]. However, the existing methods couldn't perform well in the context of intra-socket pressure measurements. Amputee gait phase recognition based on interfacial pressure remains challenging, as the multi-position pressure signals are more complex and less straightforward compared to data from conventional sensor measurements. Natural differences such as the subject's weight, socket size, and walking habits can also affect pressure behaviors.

4.3.1 Current gait phase recognition methods

Conventional gait phase recognition methods usually rely on expert knowledge to reveal the fundamental relationships in gait events. A threshold-based time-frequency analysis method is proposed in [98] for non-disabled human gait phase detection. Maqbool *et al.* [99] developed a heuristic rule-based amputee gait event detection system using the IMU data. Although these methods report high recognition accuracy, they rely heavily on extensive prior knowledge and labeled data for empirical rule-based decisions, restricting their effectiveness with unencountered samples.

With the development of machine learning technology, artificial neural network-based (ANN) approaches have been applied to gait phase detection. Among these ANN models, the Long Short-term Memory (LSTM) network is a common choice since they are well-suited for sequential data with temporal dependencies. Cai *et al.* [100] present an EMG-based gait phase recognition method using a variation of the LSTM model. Unfortunately, their method requires healthy subjects so that lower limb surface EMG signals can be obtained, which couldn't be applied to transfemoral amputees. Ding *et al.* [101] propose a gait phase detection method using a single IMU and an LSTM-based algorithm that can achieve 91.4% accuracy. Tran *et al.* [102] develop a multi-model LSTM for learning the temporal features of gait using an IMU sensor. They employ six LSTMs, each for processing one channel of the IMU data. Sarshar *et al.* [103] propose a multiple LSTM method for IMU-based gait analysis. They trained three separate LSTMs, each for estimating one individual gait phase including Toeoff, Midswing, and Heelstrike.

Besides LSTM, HMM and its variants represent another widely used class of machine learning methods for gait pattern recognition. Attal *et al.* [104] have employed the multiple-regression Hidden Markov model (MR-HMM) using foot plantar pressure for gait phase recognition. However, their focus is on non-amputee individuals, where insole pressure sensors are required for each foot. That is hard to apply to amputees. Later on, they proposed a continuous HMM (CHMM) approach [105] based on two IMUs, which deploys a single GMM-HMM to recognize gait phases. Their approach can learn in an unsupervised manner, but the average

recognition rate of 82.47% on the individual subject is not enough for gait analysis. Other popular recognition approaches can be found in [97], but there is no specific method for amputee gait phase recognition concerning interfacial pressure.

GMM-HMM extends the HMM with better recognition ability, as each emission is modeled as a multivariate Gaussian distribution. In [106, 107], researchers employed GMM-HMM for high-accuracy hand gesture recognition and human activity recognition. Given that interfacial pressure readings and gait events can be treated as sequential observations, GMM-HMM has the potential for amputee gait phase recognition.

4.3.2 Problem definition

Gait phase recognition is a key requirement for gait analysis. A large amount of complex data on gait activities makes manual gait phase labeling time-consuming and inaccurate. The research problem then focuses on how to perform accurate amputee gait phase recognition using intra-socket pressure measurement.

4.3.2.1 Intra-socket pressure measurement

Since the problem is based on intra-socket pressure measurement, the interfacial pressure is the necessary feature as input and the data collection is described below.

Fig. 4.22 illustrates the acquisition of intra-socket pressure from multiple sensors mounted on the inner surface of the prosthetic socket. These sensors can be individually placed or connected in the form of a strip or a pad. We show an example of 21 sensors on four strips arranged alongside four socket edges. They can measure pressure between the stump (residual limb protected by liner) and the socket. Each input feature frame represents the reading of all pressure sensors from the same sampling time, then the interfacial pressure is treated as multivariate sequential data. From this, we can observe the cyclical response, which indicates the gait cycles.

External sensors for gait monitoring may cause inconvenience to amputees and prosthetists during clinical tests. Aligning such gait information with current pressure measurements needs extra processing overhead. Therefore, we exclusively rely on the intra-socket pressure for gait phase recognition.

4.3.3 Gait phase description

For amputees, the gait analysis focuses on the periodic leg movement of the amputation side, namely gait cycles. The fundamental division includes Swing (SW) and Stance (ST), and there are different granularity of gait phases [108]. To keep generality and synchronize with the clinical assessment, a simplified four-phase model from the classic gait terms [109] is selected, involving Toeoff, Midswing, Heelstrike, and Midstance. **Fig.** 4.23 shows the four gait phases and the corresponding pressure curves within two consecutive gait cycles.

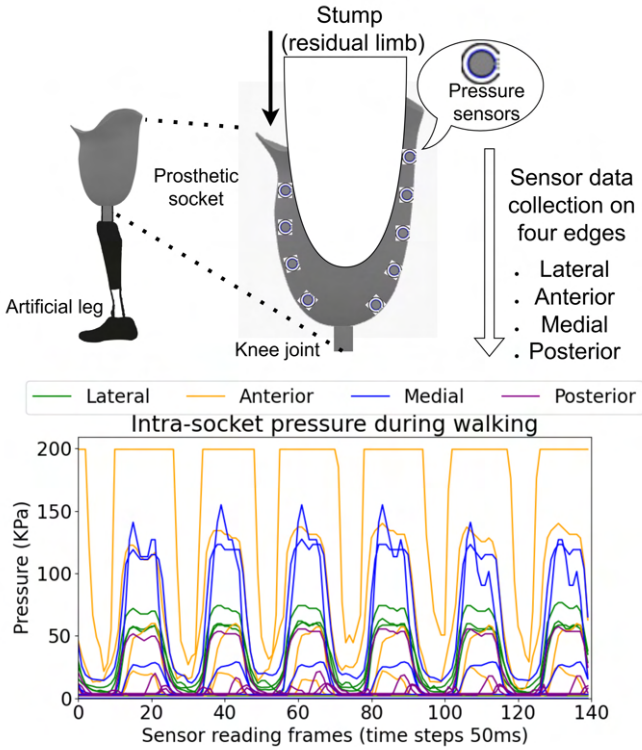


Figure 4.22: Intra-socket pressure data collection.

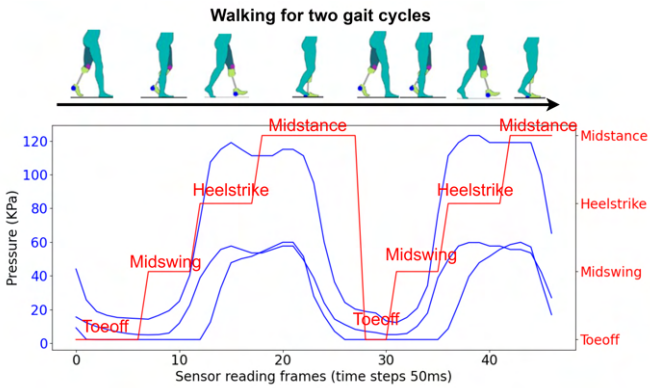


Figure 4.23: The example of four gait phases in the gait cycle.

In particular, Toeoff is the pre-swing state, where the toes leave the ground.

Midswing occurs when the foot swings in the air. Heelstrike means the heel is in contact with the ground. During Midstance, the body is supported only by the prosthetic leg. Since the gait phases are not equal in length, our approach should be able to recognize not only long phases but also short phases such as Toeoff.

In our study, we aim to match the pressure sensor readings with actual gait activities accurately. Especially, intra-socket pressure data are given as input that each pressure segment contains I data frames $X = \{x_1, \cdots, x_I\}$. The recognition task is to label them with one of the four gait phases, $L(x_i) \in \{$Toeoff, Midswing, Heelstrike, Midstance$\}$, for all $x_i \in X$.

4.3.4 Multiple GMM-HMM method

We use multiple GMM-HMM for amputee gait phase recognition. **Fig.** 4.24 shows the proposed method overview. Since the recognition target is four gait phases, the number of GMM-HMM models is set to four. The idea is to learn separate models specified for each gait phase. It runs in a supervised manner. The pressure data are partitioned into the train set and test set. The gait cycles are segmented according to gait phases to create the dataset with ground truth, which serves as the model input.

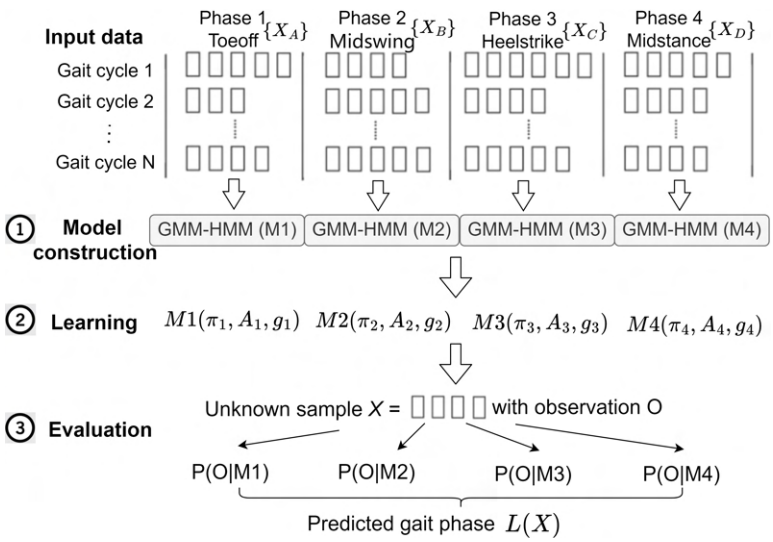


Figure 4.24: Overview of multiple GMM-HMM for gait phase recognition.

Three main processes in **Fig.** 4.24 are as follows.

1. Model construction: The structure of GMM-HMM is determined by the physical condition of the gait movement. The model parameters $\lambda = (\pi, A, g)$ are initialized based on the input pressure from the train set.

2. Learning: The parameters λ are updated by Viterbi training. The learning step iterates until the model converges.

3. Evaluation: The emission probabilities are obtained on the test dataset from the learned GMM-HMM to predict gait phases. The predicted gait phases are compared with ground truth to find the recognition accuracy.

For each GMM-HMM, the procedure is the same except that they use different input data for learning with respect to the gait phases. The details of model construction, learning, and evaluation processes are described below.

4.3.4.1 Model construction

There are different types of HMM structures, such as the ergodic model [110] with all transitions between states enabled. The moving pattern of the prosthetic leg is one-way and the gait phase appears in the order of Toeoff, Midswing, Heelstrike, Midstance during normal gait cycles. The leg movement also occurs in successive order inside each gait phase. Therefore, a typical left-to-right HMM without state skips as shown in **Fig.** 4.25 is constructed.

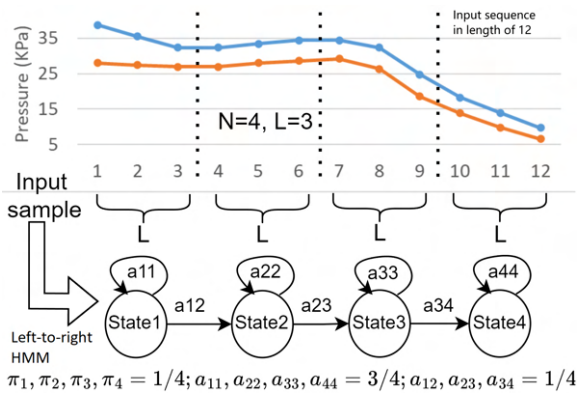


Figure 4.25: HMM model construction.

The first step in model construction is to determine the number of hidden states N and mixtures K, which is usually related to the corresponding physical significance in the patterns. In our case, since the gait cycle is already segmented in the dataset and each GMM-HMM represents one gait phase, the choice of N is determined by the length of the input pressure sequence. For example, if one input has three frames, its length is three, then N should be less or equal to three. For a statistically significant GMM-HMM, at least two hidden states are required. K depends on the observed feature distribution, which needs further search according to the input. Usually, it starts from two and ends when the accuracy rate is saturated. The observed data should be allocated to the hidden states in the model.

In a left-to-right structure, uniform state assignment is preferred. The observed sequences are uniformly divided into N hidden states with length L per state, and the Gaussian mixture model takes place in each state. **Fig.** 4.25 shows an example of a constructed model taking the input sample in the length of 12 from the Midstance phase. When N is set to 4, L is 3 for each hidden state.

The second step is to initialize GMM-HMM parameters. As discussed in the previous section, the parameters are $\lambda = (\pi, A, g)$. π and A can be derived based on the initial alignment, where each element in π is $1/N$ and the state transition matrix A is calculated by dividing the number of transitions to other states by the total number of transitions, $a_{i,i} = L/(L+1)$ and $a_{i,i+1} = 1/(L+1)$. The initialized parameters from the example input are also given in **Fig.** 4.25. The GMM parameters g are initialized by the K-means clustering algorithm, which runs ten times to maximize the likelihood.

4.3.4.2 Learning process

After model construction, the learning process updates the initialized model parameters to realign closer to the real situation. The Viterbi training is applied. It records each possible state's previous optimal path probability at each moment and the previous state of the optimal path simultaneously. Then it iterates backward continuously to find the state corresponding to the maximum probability value at the last time point and backtracks forward to get the optimal path. After that, the optimal state transition is also obtained. The transition probability A is updated according to the new allocation of observed sequences over the hidden states. g also changes with the updated mean and variance, which is used to calculate the emission probability B.

For each training iteration, this step is repeated, and new A, B can be obtained to perform the next Viterbi training. This loops until it reaches the desired iteration number, and then the learning process of GMM-HMM is completed.

4.3.5 Evaluation process

The characteristics of pressure signals are statistically captured through the learning process. As **Fig.** 4.26 shows, the evaluation process works as an isolated phase recognizer centered after the four trained GMM-HMMs, where each represents a distinct gait phase. The pressure data from the test dataset are treated as feature vectors that reflect the movement of each gait phase. Based on the learned model parameters λ, the probability computation produces the probabilistic score of the input features against each GMM-HMM.

With the learned model M with parameter λ, and the new observation O from test sequences, Viterbi decoding [111] can uncover the hidden state sequence and find the emission probability $P(O|M)$ for each of four GMM-HMM models. The final choice of gait phase X is determined by **Eq.** (4.3),

$$X = argmax(P(O|M_1), P(O|M_2), P(O|M_3), P(O|M_4)) \tag{4.3}$$

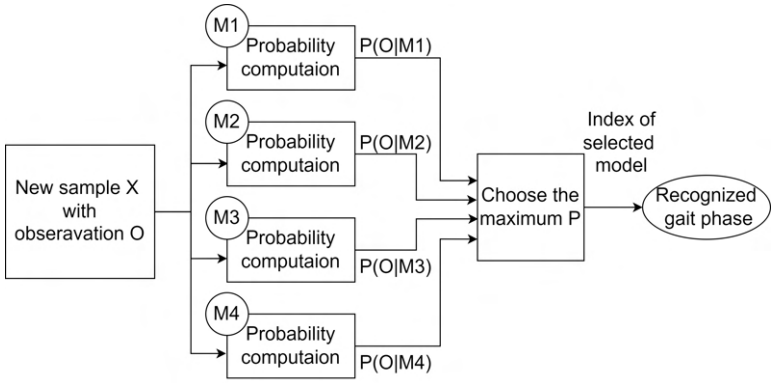


Figure 4.26: Gait phase labeling process.

where M_1 to M_4 represents GMM-HMM for gait phases 1 to 4. Eventually, the recognition decision is made on which output has the highest likelihood. Then these predicted gait phases are compared with ground truth to check the recognition rate.

4.3.6 Evaluation and results

4.3.6.1 Data collection and evaluation setup

We use data acquired from a set of clinical trials to evaluate the performance of the proposed multiple GMM-HMM approach. These clinical trials were conducted in hospitals in Spain and the UK. All trials were done with amputees after the trial protocols had been reviewed and approved by responsible national authorities [2] [3]. The trial procedure was well defined as described in [112] before the actual trials with amputees. The trial datasets are archived at [113].

An embedded pressure-measuring system [112] using wearable QTSS™ pressure sensors [62] is built for trials. Six human subjects with different profiles are involved (four males and two females; age: 58 ± 8 years; height: 174 ± 10 cm; weight: 73 ± 13 kg). All of them have transfemoral prostheses, four on the left side and two on the right side.

Pressure sensor strips are deployed on four edges inside the socket and perpendicular to the ground, including the anterior, posterior, medial, and lateral edges. The sensors on the strip are placed from the proximal end to the distal end in a fixed order with proper intervals to avoid overlapping.

[2] Comité de Ética de la Investigación con medicamentos (CEIm) and Agencia Española de Medicamentos y Productos Sanitarios (AEMPS) approvals were obtained.

[3] Health Research Authority (HRA) and Medicines and Healthcare Products Regulatory Agency (MHRA) approvals were obtained (IRAS Project ID: 292614).

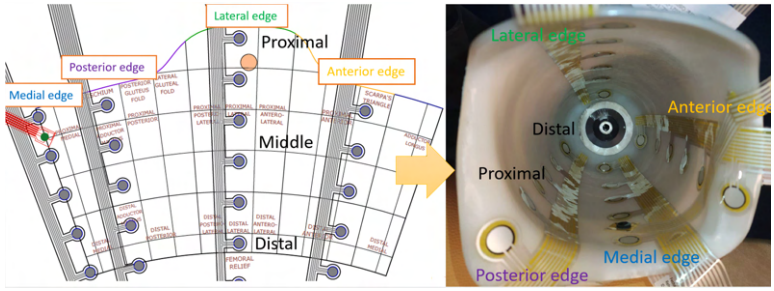


Figure 4.27: Sensor deployment map and placement in the socket.

The designed sensor layout schematic and the placement inside the socket are illustrated in **Fig.** 4.27, where the circles on the strips represent the sensing elements, and the labels indicate the region segmentation inside the socket.

Each subject wore their daily used socket and performed level ground walking for five minutes at their own speed. The measurement is taken at a rate of 20 Hz, which means 20 frames of sensor recordings per second. Besides the pressure data, an IMU is mounted on the prosthetic leg. By aligning the timestamps of the IMU data and video recording of the tests, pressure data are labeled with corresponding gait phases, which are treated as the ground truth of our dataset. We can segment the pressure frames into gait phases based on these labels. The intra-socket pressure is pre-processed to contain only valid full gait cycles, during which irregular activities other than normal walking are removed, such as turning around at the end of the walking corridor or taking short rests between gait cycles. Note that for each phase we will keep at least two frames in order to maintain a meaningful number of hidden states. After that, we count the valid gait cycles for all subjects. There are 135, 194, 144, 102, 111, and 113 complete gait cycles for Subject 1 to Subject 6, respectively.

The dataset is formatted in a list of pressure feature files, each containing a gait phase in one gait cycle. Hence there are in total four times the number of the gait cycle of feature files for one subject. The number of sensors deployed is the number of possible features observed, which varies among subjects. The evaluation is performed in three scenarios.

1. For a single subject: Separate multiple GMM-HMM models are trained and tested on each subject. 50% of the data is used in training while the other 50% is used for testing.

2. For multiple subjects: One GMM-HMM model is evaluated for multiple subjects, where the subjects' data are combined, with the same train test ratio of 50%:50%.

3. For the unseen subject: One multiple-GMM-HMM model is trained by five subjects and evaluated on the leave-out subject. The training set is 70% of

the five-subject dataset, and the test set is the whole leave-out dataset.

We use **accuracy on the test set** as the gait recognition metric, which is defined as the ratio of the number of correct gait phase labels to the total number of inputs.

4.3.6.2 Evaluation in single-subject scenario

For each subject, the accuracy results of recognized gait phases are evaluated based on different training configurations. Since the length of observed sequences for each phase has a minimum number of two, the number of hidden states is set to 2 for all GMM-HMM models, and the learning iteration is set to 30. The dataset is randomly shuffled before training.

Impact of spatial sensors In our measurement, one sensor strip is designed to hold at most 8 sensors (E1 to E8). However, the actual number of sensors deployed in the trial depends on the physical length of the subjects' sockets to avoid overlap in the bottom area.

The location of sensors is summarized in **Fig.** 4.28, where the unused sensors are marked with $\times$. P means the proximal end, M means the middle region, and D means the distal end. We can count that Subject 1 has 25 sensors, Subject 2 has 15 sensors, Subject 3 has 24 sensors, Subject 4 has 22 sensors, Subject 5 has 21 sensors, and Subject 6 has 16 sensors inside the socket. For the sensor selection, since each subject has a different sensor layout in the trial, we consider two options. One is to use the common sensor positions among all subjects, which includes the top two sensors from the proximal end of each edge. Therefore $2 \times 4 = 8$ sensors are selected, whose positions are highlighted by the red box in **Fig.** 4.28. Another assumption is that all pressure sensors used in the trial are treated as valid features, though the sensors at the distal end have a much smaller pressure variation than others.

In this situation, we traverse the Gaussian mixtures K from 2 to 6 and take the average recognition accuracy on the test set as the result shown in **Fig.** 4.29. For Subjects 1, 3, 4, 5, and 6, the 8-sensor setup gives better accuracy of 1.1% and 1.3% respectively than the all-sensor. For Subject 2, the 8-sensor has less accuracy of 2.3% than the all-sensor situation. Generally, both configuration achieves a high recognition accuracy, and 8-sensor selection works better in most cases as it extracts the more significant pressure margin, which is usually on the proximal end. However, for Subject 2, since the socket has a much shorter length than others, the selected sensors are located down to the distal region for the medial edge. The pressure behavior is different from the other five subjects. In this case, the all-sensor selection is better for Subject 2.

Selection of Gaussian mixtures K Besides the options for the spatial sensors, Gaussian mixtures can also affect the performance by finding a good fit for the input

	Subject 1				Subject 2			
Strip	**Anterior**	**Posterior**	**Medial**	**Lateral**	**Anterior**	**Posterior**	**Medial**	**Lateral**
E8	P	P	P	P	P	P	P	P
E7	P	P	M	P	P	P	D	P
E6	M	M	M	P	D	M	**X**	P
E5	M	M	M	M	**X**	D	**X**	M
E4	M	M	D	M	**X**	**X**	**X**	M
E3	D	M	**X**	M	**X**	**X**	**X**	D
E2	**X**	D	**X**	D	**X**	**X**	**X**	**X**
E1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
	Subject 3				**Subject 4**			
Strip	**Anterior**	**Posterior**	**Medial**	**Lateral**	**Anterior**	**Posterior**	**Medial**	**Lateral**
E8	P	P	P	P	P	P	P	P
E7	P	P	M	P	P	P	M	P
E6	M	M	M	P	M	M	M	P
E5	M	M	M	M	M	M	M	M
E4	M	M	D	M	D	D	D	M
E3	D	D	**X**	M	**X**	**X**	**X**	M
E2	**X**	**X**	**X**	D	**X**	**X**	**X**	D
E1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
	Subject 5				**Subject 6**			
Strip	**Anterior**	**Posterior**	**Medial**	**Lateral**	**Anterior**	**Posterior**	**Medial**	**Lateral**
E8	P	P	P	P	P	P	P	P
E7	P	P	M	P	P	P	M	P
E6	M	M	M	P	M	M	D	P
E5	M	M	D	M	D	D	**X**	M
E4	D	D	**X**	M	**X**	**X**	**X**	D
E3	**X**	**X**	**X**	M	**X**	**X**	**X**	**X**
E2	**X**	**X**	**X**	D	**X**	**X**	**X**	**X**
E1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Figure 4.28: Sensor location for six human subjects.

data. We choose the number of Gaussian mixtures K from 2 to 6 for evaluation and use the 8-sensor selection as a reference.

Fig. 4.30 plots the recognition accuracy of different Gaussian mixtures K on six subjects over all four phases. We can observe that for Subject 1, K equals 3 and 4 have better accuracy than others. For Subjects 2, 3, and 6, $K = 5$ has the best results. For Subject 4, $K = 6$ has the highest accuracy. For Subject 5, K equals 4 and 5 have the same accuracy and are higher than other choices of the mixture number.

The proper choice of K depends on the actual data length and distribution. As all of the gait phase observation sequence lengths are less than 20, it is quite enough to use 4 to 5 Gaussian mixtures to fit the emission probability of HMM. When K is larger than 5, the accuracy starts to drop. The possible reason is overfitting, as the increased model flexibility may capture unexpected pressure fluctuation rather than the underlying relationship to the gait phase.

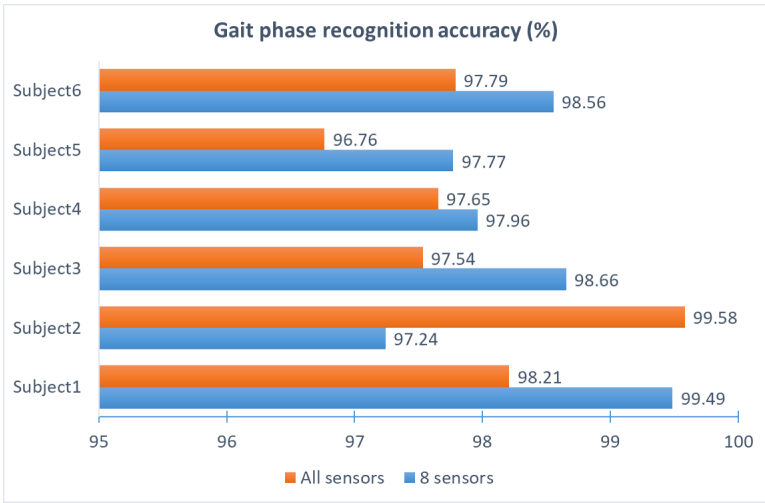
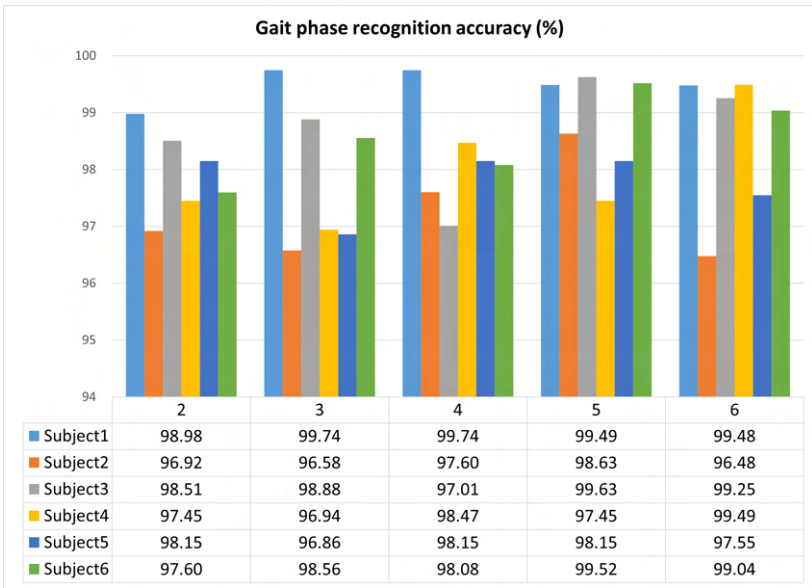


Figure 4.29: Average recognition accuracy (%) for selection on spatial sensors in single-subject scenario.



	2	3	4	5	6
Subject1	98.98	99.74	99.74	99.49	99.48
Subject2	96.92	96.58	97.60	98.63	96.48
Subject3	98.51	98.88	97.01	99.63	99.25
Subject4	97.45	96.94	98.47	97.45	99.49
Subject5	98.15	96.86	98.15	98.15	97.55
Subject6	97.60	98.56	98.08	99.52	99.04

Figure 4.30: Accuracy (%) for selection of Gaussian mixtures $K = 2$ to 6 in single-subject scenario.

4.3.6.3 Evaluation in multiple-subject scenario

Upon the single-subject scenario, we extend the gait phase recognition ability of the proposed multiple GMM-HMM method to multiple subjects. This examines the compatibility of our method with various amputees. From previous results, we consider the 8-sensor selection as the common feature. The total training set includes 400 gait cycles, and the test set includes 399 gait cycles.

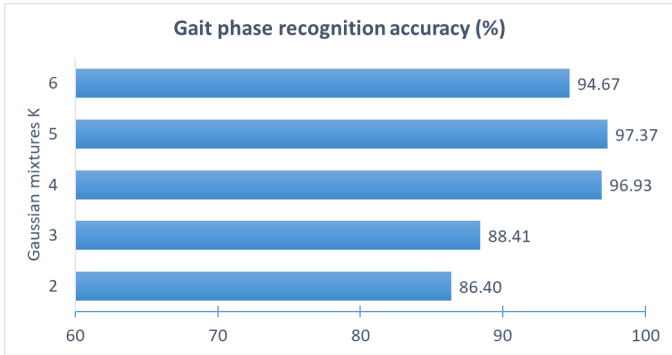


Figure 4.31: Accuracy (%) for selection of Gaussian mixtures $K = 2$ to 6 in multiple-subject scenario.

Fig. 4.31 shows the gait phase recognition accuracy among six human subjects with Gaussian mixtures K from 2 to 6. We can observe that the model with two Gaussian mixtures is not enough to fit the data, which gives a lower recognition accuracy of 2% to 11% than the model with a higher number of mixtures. The 5-mixture model can achieve fairly good accuracy on the test set for around 97.4%. This aligns with the observation in the single-subject scenario that $K = 5$ is a good choice for our interfacial pressure dataset.

To check the phase-wise accuracy, we plot the confusion matrix on recognition result for the multiple-subject scenario in **Fig.** 4.32. Here we show the proposed approach with the number of Gaussian mixtures $K = 2$ and $K = 5$.

From the $K = 2$ confusion matrix, we can see for Toeoff and Heelstrike phases, there are few recognition errors. While in the Midswing and Midstance phases, the recognition accuracy is relatively low for the 2-mixture model. There are around 23% of the testing samples in the Midswing phase are misclassified to the Toeoff phase. A similar amount of misclassifications also appear in the testing samples in the Midstance phase, which is predicted to be the Heelstrike phase. The possible reason is that when the training samples contain different human subjects, only two Gaussian mixtures per state are not enough to simulate the actual feature distribution. This can also explain why it behaves even worse in the multiple-subject scenario than in the single-subject scenario.

For the 5-mixture model, the confusion matrix shows good results with only 15 and 17 input samples misclassified for Toeoff and Midswing phases. For Heelstrike

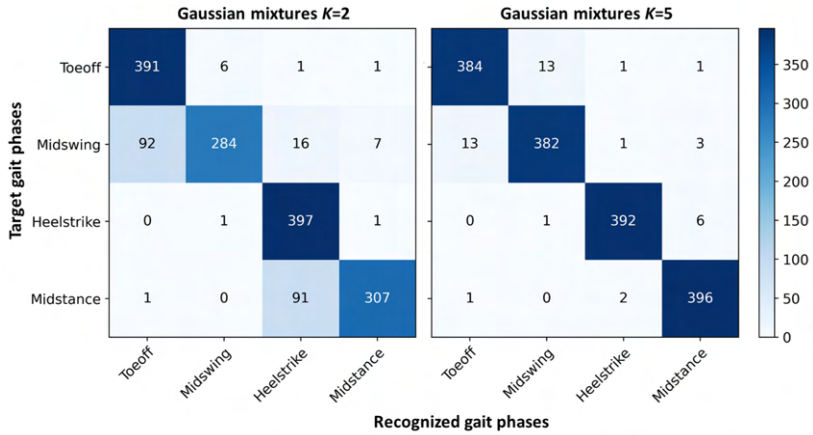


Figure 4.32: Confusion matrices for four gait phases with Gaussian mixtures $K = 2$ and $K = 5$.

and Midstance phases, the recognition rate is 98.25% and 99.25%, respectively. Since the pressure change during the first two phases in a gait cycle is normally not as obvious as the other two phases, it's harder to distinguish them with the limited number of training observations.

4.3.6.4 Evaluation in unseen-subject scenario

To better investigate the generalization of the proposed method, a leave-one-subject-out validation is designed. As the results of the previous scenarios suggest, here we set the Gaussian mixture K to 5, and use the 8-sensor selection as the common feature. **Fig.** 4.33 shows the gait phase recognition accuracy on the leave-out subject evaluated on the GMM-HMM model trained from the other five subjects. Among the six cases, the best case is for subject 3, on which the model reaches a recognition accuracy of 84.55%.

The evaluation results on unseen subjects indicate that the proposed method has the ability for generalization, but the performance drops as the data distribution can vary a lot between trained and unseen subjects. The current model's capability is limited as the available number of subjects' data for training is small. The multiple-GMM-HMM can improve its generalizability by learning from more cases with diverse conditions.

4.3.6.5 Summary

For the single-subject scenario, the supervised multiple GMM-HMM can reach around 99.5% recognition accuracy with proper parameter selection on the number of sensors and Gaussian mixtures. For the multiple-subject scenario, our approach

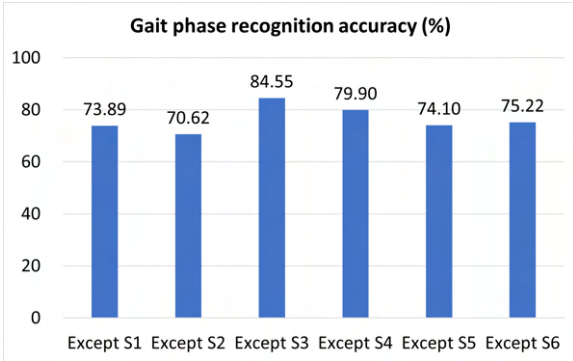


Figure 4.33: Accuracy (%) in unseen-subject scenario with Gaussian mixture $K = 5$ and 8-sensor.

can still maintain a high recognition accuracy of 97.37% with five Gaussian mixtures and an 8-sensor configuration. For the unseen-subject scenario, our model can reach an average of 76.38% recognition accuracy on the leave-out subject, which shows the potential for model generalization.

4.3.7 Comparison of the proposed approach with widely-used recognition methods

In this section, we compare the performance of our multiple GMM-HMM approach with two well-known gait phase recognition methods, encompassing the unsupervised single GMM-HMM (CHMM) method [105], and the supervised LSTM model [101].

For the single GMM-HMM method, the model construction is similar to the multiple GMM-HMM approach. However, instead of employing separate models for distinct gait phases, it represents all gait phases using one model, which means the four states in **Fig.** 4.25 are naturally linked with four physical gait phases. Therefore, it possesses the capability to discern gait phases throughout a complete gait cycle directly.

LSTM is a type of recurrent neural network model that is designed to extract time series data features for both short-term and long-term. Since the pressure sensor data are collected continuously when the subject walks with the artificial leg, the temporal pressure information can be extracted using the LSTM network. Then we can predict the next gait event based on the current pressure sequence.

4.3.7.1 Recognition accuracy comparison

We evaluate the same scenarios including the single-subject, multiple-subject, and unseen-subject using the single GMM-HMM and LSTM methods, and compare their gait phase recognition accuracy with our approach. For ease of comparison,

we choose the 8-sensor situation for all scenarios. We use the same train-test ratio to evaluate these two methods as in our approach.

For the single GMM-HMM method, we segment the original dataset by gait cycles, and each gait cycle is treated as one input sequence. Since this method is unsupervised, we define the training process as updating model parameters according to the input. Then for the test process, the model parameters are fixed to label the gait phase for every data point in the input sequence. We have tried different selections of Gaussian mixtures K, and we found that here $K = 3$ performs the best. So we use this for all evaluations on the single GMM-HMM method.

The LSTM network consists of two LSTM layers and one fully connected layer. To construct the train set and test set for LSTM, we use a fixed-length sliding window as in [102] to generate multiple subsequences from the whole walking series. The window size is set to 20, which is aligned with the pressure sensor sampling frequency of 20 Hz. The sliding step is one as we execute a one-step prediction immediately after the input subsequence. We set 64 hidden neurons in each LSTM layer and one output neuron for the fully connected layer. The final output of the last fully connected layer is rounded to represent the gait phase associated with the input sample. The model is trained with a batch size of 64 for 5 epochs on the single subject scenario and trained for 10 epochs on the other two scenarios.

Table 4.4: Gait phase recognition accuracy comparison

		Gait phase recognition accuracy (%)		
Scenarios	Subject	Multiple GMM-HMM (Ours)	Single GMM-HMM [105]	LSTM [101]
A: Single subject	S1	99.49	59.41	85.28
	S2	97.24	50.17	90.57
	S3	98.66	42.22	84.65
	S4	97.96	49.87	81.47
	S5	97.77	58.52	73.85
	S6	98.56	52.98	85.00
	Average	98.28	52.20	83.47
B: Multiple subject	All	92.76	48.30	77.10
C: Unseen subject	Ex. S1	73.89	66.67	75.28
	Ex. S2	70.62	50.16	77.38
	Ex. S3	84.55	56.16	82.34
	Ex. S4	79.90	57.23	80.99
	Ex. S5	74.10	44.76	60.25
	Ex. S6	75.22	69.86	80.28
	Average	76.38	57.47	76.09

Tab. 4.4 summarize the evaluation results for all three scenarios using these

recognition methods. For the single-subject scenario, our approach outperforms the single GMM-HMM and LSTM by 46.1% and 14.8% on average in terms of gait phase recognition accuracy. In the multiple-subject context, despite observing a decrease in the overall accuracy when contrasted with the single-subject scenario, the multiple GMM-HMM approach still yields superior performance than the single GMM-HMM and LSTM by 44.5% and 15.7% of accuracy respectively. In the context of unseen subjects, both LSTM and our approach demonstrate comparable accuracy, with an average of 76%. The single GMM-HMM method shows notably inferior accuracy results than these two methods. Therefore, the performance of our multiple GMM-HMM approach is better than the unsupervised single GMM-HMM method in all evaluated situations. Compared with the supervised LSTM model, the proposed approach is better in single-subject and multiple-subject scenarios, while remaining competitive in the unseen-subject tests.

4.3.7.2 Model complexity comparison

Besides the model recognition performance, the model complexity is also a concern for its implementation on embedded platforms. Many applications for accurate amputee gait phase recognition such as exoskeleton movement control [94] and automatic prosthesis socket size adjustment [95, 96], in which real-time recognition is preferred. Here we discuss the inference computation complexity and model size of our approach and the aforementioned recognition methods.

For the single GMM-HMM model, the inference time complexity can be written as $O(TNK + TN^2)$, where T is the length of the input sequence, N is the number of hidden states, and K is the number of Gaussian mixtures. N, K can be treated as constants, for example, $N = 4$, $K = 3$. So its execution time grows linear with the input sequence length T.

Since the proposed multiple GMM-HMM approach is a combination of n single GMM-HMM models with a comparator at the end, the computation complexity can be approximated as $O(n \times (TNK + TN^2))$. In our case, n equals 4 as there are four gait phases to recognize. In the multiple GMM-HMM approach, T represents the length of one gait phase instead of the whole gait cycle, which is smaller than that in the single GMM-HMM method, and typically in the range of 5 to 20. Therefore, our approach also has the linear time complexity to the input sequence length T, and the actual computation time is slightly larger than the single GMM-HMM.

On the other hand, the LSTM method requires $O(TH^2)$, where T is the length of the input sequence, and H is the number of neurons in the LSTM layer. Normally, H is much larger than N, for example, we have $T = 20$ and use $H = 64$ in our evaluation. Moreover, our complete LSTM model comprises two LSTM layers and one fully connected layer, which largely increases the computation time. So regarding the inference time, GMM-HMM-based methods are more efficient than the LSTM method.

To compare the model size, we count the model parameters for three methods. In the single GMM-HMM model, the total number of parameters is $N^2 + K \times$

$(1 + D + D^2)$, where N, K represents the states and Gaussian mixtures, and D is the dimensions of the input data. Considering the 8-sensor situation, we have the model size of $4^2 + 3 \times (1 + 8 + 8^2) = 235$ parameters. Similarly, we can calculate the model size of our multiple GMM-HMM approach as 940 parameters when we keep the same N, K, and D.

For the LSTM model, the first LSTM layer comprises $4 \times (64 + 8) \times 64 = 18432$ weights and 256 biases. The second LSTM layer consists of 33024 parameters and the fully connected layer has 65 parameters. Thus, the entire LSTM model encompasses around 52K parameters, whose size is 55 times larger than that of the proposed approach.

In this case, our multiple GMM-HMM approach has better model complexity regarding both gait recognition time and model size than the LSTM approach. Though it requires more computation time and storage than the single GMM-HMM method, the enlarged resources are affordable for most embedded devices, while the gait phase recognition accuracy can be improved a lot.

4.4 Summary

In this chapter, a wearable pressure measurement system is developed for collecting and analyzing dynamic operation conditions intra-socket for lower-limb amputees. The feasibility of deploying clustering algorithms on such embedded sensor system is also evaluated. To address the need for efficient data analysis in designing more comfortable prosthetic sockets, we propose two closely related applications that aim to leverage advanced machine learning techniques.

In the first application, we present a case study that aims to improve the sensor deployment by reducing the redundancy in local areas. A clustering-based analysis method is proposed to evaluate the sensor density and give references to the removal of sensors. The proposed method can be useful in socket rectification and smart prosthetic socket design. It assists researchers in selecting efficient pressure sensor positions inside sockets. The JSD evaluation result and the mean pressure comparison show the acceptable loss in the pressure features after redundancy removal, which validates the efficacy of our method. To investigate further, this method can be applied to various patients and multiple sensor deployments. Currently, this is limited by our experimental conditions. We are also aware of the possible need to increase sensor density for higher coverage in some critical regions, which still needs to be identified.

In the second application, we present the multiple GMM-HMM approach for lower limb amputee gait phase recognition based on the intra-socket pressure measurement. The proposed method can learn the gait pattern from interfacial pressure features collected by wearable sensors. It is validated on a dataset containing six human subjects. The evaluation results on the trained models show satisfying accuracy. Relatively good recognition accuracy for leave-one-out validation proves the generalization ability of our method. This approach shows superior recogni-

tion accuracy in all scenarios when compared to the single GMM-HMM method, and in single-subject and multiple-subject scenarios compared to LSTM. Our approach requires much fewer computation resources but still maintains recognition accuracy advantages in the context of unseen subjects compared with the LSTM method. The future work aims to integrate the recognition process in gait analysis for comfort socket design. One essential step is automatic gait recognition on edge devices in real-time. Also, more subjects could be included to support the general applicability and further extend the recognition ability of the proposed approach.

Chapter 5

Concluding remark

5.1 Summary

This thesis has addressed key challenges in efficient machine learning for edge computing by advancing both architectural and application-level solutions. A major contribution is the development of a cycle-accurate NoC-DNN simulator, designed to explore the extensive design space of NoC-based DNN accelerators. By simulating critical performance metrics such as latency, resource utilization, and traffic patterns, the simulator enables in-depth evaluations of various design parameters, including NoC size, task mapping strategies, and memory controller placement. This tool provides a foundation for optimizing data traffic management, distributing tasks, and improving the scalability of hardware systems without the need for costly prototyping. Its flexibility and accuracy make it a valuable resource for researchers developing next-generation machine learning accelerators for edge devices.

In addition to the simulation tool, novel NoC-based architectural designs are introduced that leverage in-network processing to address the limitations of traditional NoC-DNNs. By integrating computation directly into the network devices, these designs reduce data movement and improve overall system efficiency. Techniques such as activation in network and pooling on-the-go demonstrate how localized processing within the NoC can achieve lower latency for DNN inference. These innovations enhance the efficiency and adaptability of NoC-based accelerators, making them suitable for a wide range of DNN workloads with varying computational demands.

On the application side, two machine learning approaches tailored to embedded sensor systems are presented, particularly for comfortable prosthetic socket design aimed at lower-limb amputees. A wearable pressure measurement system is built for collecting operational data of prosthetic sockets. Then a clustering-based approach is proposed to optimize sensor placement by identifying and eliminating redundancy in deployment, leading to more efficient and effective pressure sensing configurations. This method provides practical insights into socket rectification

and customization while maintaining the accuracy of pressure data collection. Additionally, a multiple GMM-HMM-based gait phase recognition approach is proposed. This method uses interfacial pressure data to analyze and classify amputee gait patterns, which demonstrates superior accuracy and computational efficiency compared to traditional methods, highlighting its potential for real-time applications in wearable systems. These contributions collectively advance the state of the art for machine learning on edge by addressing both hardware and application challenges.

5.2 Future work

Several directions can be extended upon the foundation established in this thesis as future work that aim to address emerging challenges and opportunities in both NoC-based accelerator designs and machine learning applications for embedded systems.

On the architectural side, future work will focus on enhancing the capabilities of NoC-based DNN accelerators. One promising direction is the integration of in-network processing techniques on more operations, such as MAC and normalization. Adaptive dataflow and context-aware computation within the NoC are also worth investigation. These enhancements could further reduce latency and energy consumption by dynamically adjusting to workload characteristics. Additionally, exploring the scalability of these architectures for increasingly complex DNN models, such as transformer-based networks, is critical. This requires the development of more sophisticated simulation frameworks to evaluate performance under diverse scenarios.

In terms of the simulation tool, expanding the functionality of the NoC-DNN simulator is an essential next step. This includes support for more complex network topologies, such as hierarchical or hybrid NoC designs, and the ability to simulate heterogeneous workloads combining DNN inference with other edge tasks. Moreover, incorporating energy/area modeling into the simulator would provide practical estimations and deeper insights into the trade-offs between performance and power/area efficiency, aiding in the design of more sustainable edge computing systems.

Lastly, for machine learning applications in embedded sensor systems, future work will focus on extending the proposed methodologies to a broader range of use cases and environments. For example, the clustering-based sensor deployment optimization can be validated across different sensor configurations to include applications beyond prosthetic sockets. Similarly, the gait phase recognition approach could be generalized to accommodate a wider variety of subjects and gait patterns in various activities. Integrating these applications into real-time edge processing systems will be a critical step, requiring efficient hardware-software co-design to achieve low-latency and high-reliability performance.

References

[1] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[6] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[7] S. M. Nabavinejad, M. Baharloo, K.-C. Chen, M. Palesi, T. Kogel, and M. Ebrahimi, "An overview of efficient interconnection networks for deep neural network accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 3, pp. 268–282, 2020.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[10] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 12.6," 2024.

[11] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 26–35, 2016.

[12] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, IEEE, 2017.

[13] C. Zhuge, X. Liu, X. Zhang, S. Gummadi, J. Xiong, and D. Chen, "Face recognition with hybrid efficient convolution algorithms on FPGAs," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 123–128, 2018.

[14] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[15] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[16] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2820–2828, 2019.

[17] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[18] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 92–104, 2015.

[19] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[20] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[21] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[22] K.-C. Chen, M. Ebrahimi, T.-Y. Wang, and Y.-C. Yang, "NoC-based DNN accelerator: A future design paradigm," in *Proceedings of 13th IEEE/ACM international symposium on networks-on-chip*, pp. 1–8, 2019.

[23] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, pp. 365–376, 2011.

[24] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers Inc., 2004.

[25] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 1, pp. 1–28, 2010.

[26] K. T. Johnson, A. R. Hurson, and B. Shirazi, "General-purpose systolic arrays," *Computer*, vol. 26, no. 11, pp. 20–31, 1993.

[27] X. Liu, W. Wen, X. Qian, H. Li, and Y. Chen, "Neu-noc: A high-efficient interconnection network for accelerated neuromorphic systems," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 141–146, IEEE, 2018.

[28] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[29] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[30] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–27, 2019.

[31] J. Yang, H. Zheng, and A. Louri, "Venus: A versatile deep neural network accelerator architecture design for multiple applications," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2023.

[32] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.

[33] S. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, "SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–26, 2020.

[34] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *Proceedings of 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[35] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[36] K.-C. J. Chen, M. Ebrahimi, T.-Y. Wang, Y.-C. Yang, and Y.-H. Liao, "A NoC-based simulator for design and evaluation of deep neural networks," *Microprocessors and Microsystems*, vol. 77, p. 103145, 2020.

[37] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "Stonne: Enabling cycle-level microarchitectural simulation for DNN inference accelerators," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pp. 201–213, 2021.

[38] K.-C. J. Chen, T.-Y. G. Wang, and Y.-C. A. Yang, "Cycle-accurate noc-based convolutional neural network simulator," in *Proceedings of IEEE International Conference on Omni-Layer Intelligent Systems*, pp. 199–204, 2019.

[39] L. Ke, X. He, and X. Zhang, "Nnest: Early-stage design space exploration tool for neural network inference accelerators," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[40] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *Proceedings of IEEE 26th International Conference on Application-specific Systems, Architectures and Processors*, pp. 162–163, 2015.

[41] K.-C. Chen and Y.-S. Liao, "A reconfigurable deep neural network on chip design with flexible convolutional operations," in *Proceedings of 15th International Workshop on NoC Architectures*, pp. 1–5, 2022.

[42] S. Abadal, A. Mestres, R. Martinez, E. Alarcon, A. Cabellos-Aparicio, and R. Martinez, "Multicast on-chip traffic analysis targeting manycore NoC design," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 370–378, 2015.

[43] Y. Ouyang, F. Tang, C. Hu, W. Zhou, and Q. Wang, "MMNNN: A tree-based multicast mechanism for NoC-based deep neural network accelerators," *Microprocessors and Microsystems*, vol. 85, p. 104242, 2021.

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[45] B. Wang and Z. Lu, "Flexible and efficient QoS provisioning in AXI4-based network-on-chip architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1523–1536, 2021.

[46] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 33–42, 2009.

[47] H. Ahmadinejad, F. Refan, and H. S. Sarjoughian, "NoC simulation modeling in DEVS-suite," in *Proceedings of Symposium on Theory of Modeling and Simulation: DEVS Integrative M&S Symposium*, pp. 134–139, 2011.

[48] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7263–7271, 2017.

[49] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, *et al.*, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, 2010.

[50] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2018.

[51] J. Janesky, N. Rizzo, D. Houssameddine, R. Whig, F. Mancoff, M. DeHerrera, J. Sun, M. Schneider, H. Chia, S. Aggarwal, *et al.*, "Device performance in a fully functional 800MHz DDR3 spin torque magnetic random access memory," in *Proceedings of 5th IEEE International Memory Workshop*, pp. 17–20, 2013.

[52] L. Zhu, W. Fan, C. Dai, S. Zhou, Y. Xue, Z. Lu, L. Li, and Y. Fu, "A NoC-based spatial DNN inference accelerator with memory-friendly dataflow," *IEEE Design&Test*, 2023.

[53] Z. Lu, "PiN: Processing in Network-on-Chip," *IEEE Design&Test*, 2023.

[54] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.

[55] X. Hu, H. Huang, X. Li, X. Zheng, Q. Ren, J. He, and X. Xiong, "High-performance reconfigurable DNN accelerator on a bandwidth-limited embedded system," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 6, pp. 1–20, 2022.

[56] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[57] L. Zhu, W. Fan, C. Dai, S. Zhou, Y. Xue, Z. Lu, L. Li, and Y. Fu, "A NoC-based spatial DNN inference accelerator with memory-friendly dataflow," *IEEE Design & Test*, 2023.

[58] S. K. Mandal, A. Krishnakumar, R. Ayoub, M. Kishinevsky, and U. Y. Ogras, "Performance analysis of priority-aware NoCs with deflection routing under traffic congestion," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–9, 2020.

[59] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR*, 2015.

[60] L. Paternò, M. Ibrahimi, E. Gruppioni, A. Menciassi, and L. Ricotti, "Sockets for limb prostheses: a review of existing technologies and open challenges," *IEEE Transactions on Biomedical Engineering*, vol. 65, no. 9, pp. 1996–2010, 2018.

[61] S.-T. Ko, F. Asplund, and B. Zeybek, "A scoping review of pressure measurements in prosthetic sockets of transfemoral amputees during ambulation: key considerations for sensor design," *Sensors*, vol. 21, no. 15: 5016, 2021.

[62] V. Dejke, M. P. Eng, K. Brinkfeldt, J. Charnley, D. Lussey, and C. Lussey, "Development of prototype low-cost QTSS™ wearable flexible more enviro-friendly pressure, shear, and friction sensors for dynamic prosthetic fit monitoring," *Sensors*, vol. 21, no. 11: 3764, 2021.

[63] freeRTOS.org, "Tensilica xtensa customizable processors freeRTOS demo." `https://www.freertos.org/index.html` [Accessed: Dec, 2020].

[64] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, (Berkeley, Calif.), pp. 281–297, 1967.

[65] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[66] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and information systems*, vol. 7, no. 3, pp. 358–386, 2005.

[67] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, P. Yan, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML, "The UCR Time Series Classification Archive," October 2018. `https://www.cs.ucr.edu/~eamonn/time$_$series$_$data$_$2018/`.

[68] S. Gupta, K. J. Loh, and A. Pedtke, "Sensing and actuation technologies for smart socket prostheses," *Biomedical engineering letters*, vol. 10, no. 1, pp. 103–118, 2020.

[69] E. A. Al-Fakih, N. A. Abu Osman, and F. R. Mahmad Adikan, "Techniques for interface stress measurements within prosthetic sockets of transtibial amputees: A review of the past 50 years of research," *Sensors*, vol. 16, no. 7, 2016.

[70] D.-J. Chang, A. H. Desoky, M. Ouyang, and E. C. Rouchka, "Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu," in *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pp. 501–506, 2009.

[71] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 145–151, 1991.

[72] S. Commuri, J. Day, C. P. Dionne, and W. J. Ertl, "Assessment of pressures within the prosthetic socket of a person with osteomyoplastic amputation during varied walking tasks," *JPO: Journal of Prosthetics and Orthotics*, vol. 22, no. 2, pp. 127–137, 2010.

[73] V.-T. Tran, T. Kenta, A. Hanafusa, S.-i. Yamamoto, K. Ohnishi, H. Otsuka, and Y. Agarie, "Analyzing the pressure and shear stress of contact interface inside the trans-femoral socket during walking," *SEATUC journal of science and engineering*, vol. 1, no. 1, pp. 104–109, 2020.

[74] S. Ali, N. A. A. Osman, N. Mortaza, A. Eshraghi, H. Gholizadeh, and W. A. B. B. Wan Abas, "Clinical investigation of the interface pressure in the transtibial socket with dermo and seal-in x5 liner during walking and their effect

on patient satisfaction," *Clinical Biomechanics*, vol. 27, no. 9, pp. 943–948, 2012.

[75] S. Ali, N. A. Abu Osman, A. Eshraghi, H. Gholizadeh, N. A. bin Abd razak, and W. A. B. B. Wan Abas, "Interface pressure in transtibial socket during ascent and descent on stairs and its effect on patient satisfaction," *Clinical Biomechanics*, vol. 28, no. 9, pp. 994–999, 2013.

[76] F. Jasni, N. A. Hamzaid, A. G. A. Muthalif, Z. Zakaria, H. N. Shasmin, and S.-C. Ng, "In-socket sensory system for transfemoral amputees using piezoelectric sensors: An efficacy study," *IEEE/ASME Transactions on Mechatronics*, vol. 21, no. 5, pp. 2466–2476, 2016.

[77] P. D'Urso, D. Di Lallo, and E. A. Maharaj, "Autoregressive model-based fuzzy clustering and its application for detecting information redundancy in air pollution monitoring networks," *Soft Computing*, vol. 17, no. 1, pp. 83–131, 2013.

[78] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.

[79] F. S. Frillici and F. Rotini, "Prosthesis socket design through shape optimization," *Computer-Aided Design and Applications*, vol. 10, no. 5, pp. 863–876, 2013.

[80] G. Jain, T. Mahara, and K. N. Tripathi, *A Survey of Similarity Measures for Collaborative Filtering-based Recommender System*, pp. 343–352. Springer, 2020.

[81] A. S. Shirkhorshidi, S. Aghabozorgi, and T. Y. Wah, "A comparison study on similarity and dissimilarity measures in clustering continuous data," *PloS one*, vol. 10, no. 12, p. e0144059, 2015.

[82] F. Nielsen, "On a generalization of the jensen–shannon divergence and the jensen–shannon centroid," *Entropy*, vol. 22, no. 2, 2020.

[83] Novel Electronics, "Novel sensor s2006." `http://www.novelelectronics.de/novelcontent/sensors` [Accessed: Mar, 2021].

[84] Össur, "Direct socket tf." `https://www.ossur.com/en-gb/prosthetics/sockets/direct-socket-tf`[Accessed: Jan, 2022].

[85] E. S. Neumann, J. S. Wong, and R. L. Drollinger, "Concepts of pressure in an ischial containment socket: Measurement," *JPO: Journal of Prosthetics and Orthotics*, vol. 17, no. 1, pp. 2–11, 2005.

[86] Novel Electronics, "Pliance-rls prothesis: Pressure between human and prosthesis.." `https://www.novel.de/products/pliance/prosthesis/` [Accessed: Mar, 2021].

[87] D. Chen, P. Su, S. Ottikkutti, P. Vartholomeos, K. N. Tahmasebi, and M. Karamousadakis, "Analyzing dynamic operational conditions of limb prosthetic sockets with a mechatronics-twin framework," *Applied Sciences*, vol. 12, no. 3, p. 986, 2022.

[88] D. Dong, C. Ma, M. Wang, H. T. Vu, B. Vanderborght, and Y. Sun, "A low-cost framework for the recognition of human motion gait phases and patterns based on multi-source perception fusion," *Engineering Applications of Artificial Intelligence*, vol. 120, p. 105886, 2023.

[89] T. Lencioni, I. Carpinella, M. Rabuffetti, A. Marzegan, and M. Ferrarin, "Human kinematic, kinetic and EMG data during different walking and stair ascending and descending tasks," *Scientific Data*, vol. 6, no. 1, p. 309, 2019.

[90] U. Trinler, K. Hollands, R. Jones, and R. Baker, "A systematic review of approaches to modeling lower limb muscle forces during gait: Applicability to clinical gait analyses," *Gait & Posture*, vol. 61, pp. 353–361, 2018.

[91] Z. Zhu, P. Su, S. Zhong, J. Huang, S. Ottikkutti, K. N. Tahmasebi, Z. Zou, L. Zheng, and D. Chen, "Using a VAE-SOM architecture for anomaly detection of flexible sensors in limb prosthesis," *Journal of Industrial Information Integration*, vol. 35, p. 100490, 2023.

[92] T. Yan, M. Cempini, C. M. Oddo, and N. Vitiello, "Review of assistive strategies in powered lower-limb orthoses and exoskeletons," *Robotics and Autonomous Systems*, vol. 64, pp. 120–136, 2015.

[93] B. Kalita, J. Narayan, and S. K. Dwivedy, "Development of active lower limb robotic-based orthosis and exoskeleton devices: a systematic review," *International Journal of Social Robotics*, vol. 13, pp. 775–793, 2021.

[94] M. K. Ishmael, D. Archangeli, and T. Lenzi, "Powered hip exoskeleton improves walking economy in individuals with above-knee amputation," *Nature Medicine*, vol. 27, no. 10, pp. 1783–1788, 2021.

[95] E. J. Weathersby, J. L. Garbini, B. G. Larsen, J. B. McLean, A. C. Vamos, and J. E. Sanders, "Automatic control of prosthetic socket size for people with transtibial amputation: Implementation and evaluation," *IEEE Transactions on Biomedical Engineering*, vol. 68, no. 1, pp. 36–46, 2020.

[96] J.-H. Seo, H.-J. Lee, D.-W. Seo, D.-K. Lee, O.-W. Kwon, M.-K. Kwak, and K.-H. Lee, "A prosthetic socket with active volume compensation for amputated lower limb," *Sensors*, vol. 21, no. 2, p. 407, 2021.

[97] H. T. T. Vu, D. Dong, H.-L. Cao, T. Verstraten, D. Lefeber, B. Vanderborght, and J. Geeroms, "A review of gait phase detection algorithms for lower limb prostheses," *Sensors*, vol. 20, no. 14, p. 3972, 2020.

[98] S. Khandelwal and N. Wickström, "Gait event detection in real-world environment for long-term applications: Incorporating domain knowledge into time-frequency analysis," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 24, no. 12, pp. 1363–1372, 2016.

[99] H. F. Maqbool, M. A. B. Husman, M. I. Awad, A. Abouhossein, N. Iqbal, M. Tahir, and A. A. Dehghani-Sanij, "Heuristic real-time detection of temporal gait events for lower limb amputees," *IEEE Sensors Journal*, vol. 19, no. 8, pp. 3138–3148, 2018.

[100] S. Cai, D. Chen, B. Fan, M. Du, G. Bao, and G. Li, "Gait phases recognition based on lower limb sEMG signals using LDA-PSO-LSTM algorithm," *Biomedical Signal Processing and Control*, vol. 80, p. 104272, 2023.

[101] Z. Ding, C. Yang, K. Xing, X. Ma, K. Yang, H. Guo, C. Yi, and F. Jiang, "The real time gait phase detection based on Long Short-term Memory," in *Proceedings of 3rd IEEE International Conference on Data Science in Cyberspace (DSC)*, pp. 33–38, 2018.

[102] L. Tran, T. Hoang, T. Nguyen, H. Kim, and D. Choi, "Multi-model long short-term memory network for gait recognition using window-based data segment," *IEEE Access*, vol. 9, pp. 23826–23839, 2021.

[103] M. Sarshar, S. Polturi, and L. Schega, "Gait phase estimation by using LSTM in IMU-based gait analysis—Proof of concept," *Sensors*, vol. 21, no. 17, p. 5749, 2021.

[104] F. Attal, Y. Amirat, A. Chibani, and S. Mohammed, "Automatic recognition of gait phases using a multiple-regression hidden Markov model," *IEEE/ASME Transactions on Mechatronics*, vol. 23, no. 4, pp. 1597–1607, 2018.

[105] F. Attal, Y. Amirat, A. Chibani, and S. Mohammed, "Human gait phase recognition using a hidden Markov model framework," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10299–10304, 2020.

[106] J. Yang, J. Pan, and J. Li, "sEMG-based continuous hand gesture recognition using gmm-hmm and threshold model," in *Proceedings of IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1509–1514, 2017.

[107] X. Cheng, B. Huang, and J. Zong, "Device-free human activity recognition based on gmm-hmm using channel state information," *IEEE Access*, vol. 9, pp. 76592–76601, 2021.

[108] J. Taborri, E. Palermo, S. Rossi, and P. Cappa, "Gait partitioning methods: A systematic review," *Sensors*, vol. 16, no. 1: 66, 2016.

[109] T. Castermans, M. Duvinage, G. Cheron, and T. Dutoit, "Towards effective non-invasive brain-computer interfaces dedicated to gait rehabilitation systems," *Brain Sciences*, vol. 4, no. 1, pp. 1–48, 2013.

[110] S. S. Kumar and V. Ramasubramanian, "Automatic language identification using ergodic-hmm," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, pp. 609–612, 2005.

[111] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[112] Z. Lu, W. Zhu, Y. Chen, J. Charnley, V. Dejke, A. Pomazanskyi, S.-T. Ko, B. Zeybek, P. Mehryar, Z. Ali, *et al.*, "Wearable pressure sensing for lower limb amputees," in *Proceedings of IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 105–109, 2022.

[113] The SocketSense Consortium, "SocketSense Open Access Data," Dec. 2022.