Distributed Optimization and Machine Learning for Virtualized 6G Wireless Networks

Ali Ehsanian

▶ To cite this version:

Ali Ehsanian. Distributed Optimization and Machine Learning for Virtualized 6G Wireless Networks. Networking and Internet Architecture [cs.NI]. Sorbonne Université, 2024. English. NNT : 2024SORUS406 . tel-04910728

HAL Id: tel-04910728
https://theses.hal.science/tel-04910728v1

Submitted on 24 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed Optimization and Machine Learning for Virtualized 6G Wireless Networks

Dissertation

submitted to

Sorbonne Université

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Author:

Ali Ehsanian

Scheduled for defense on October 15, 2024, before a committee composed of:

Reviewers

Prof.	**Christos Verikoukis**	University of Patras, Greece
Prof.	**Ferran Adelantados**	Universitat Oberta de Catalunya, Spain

Examiners

Prof.	**Maria ZULUAGA (president of the jury)**	Eurecom, France
Prof.	**Dionysis Xenakis**	NKUA, Greece

Thesis Advisor

Prof.	**Thrasyvoulos Spyropoulos**	Eurecom, France

SORBONNE UNIVERSITÉ

EURECOM
Sophia Antipolis

Distribuée et Apprentissage Automatique pour les Réseaux sans fil Virtualisés 6G

Thèse

soumise à

Sorbonne Université

pour l'obtention du Grade de Docteur

présentée par:

Ali Ehsanian

Soutenance de thèse prévue le 15 *octobre* 2024 *devant le jury composé de:*

Rapporteur

Prof.	**Christos Verikoukis**	University of Patras, Grèce
Prof.	**Ferran Adelantado**	Universitat Oberta de Catalunya, Espagne

Examinateur

Prof.	**Maria ZULUAGA (présidente du jury)**	Eurecom, France
Prof.	**Dionysis Xenakis**	NKUA, Grèce

Directeur de Thèse

Prof.	**Thrasyvoulos Spyropoulos**	Eurecom, France

Abstract

Network slicing has emerged as a transformative concept in the evolution of 5G networks, marking a significant shift in how network resources are managed and optimized. It enables network operators to partition their physical infrastructure, from the network edge to the data center, into multiple virtual slices. Each slice can be tailored to meet the specific demands of different tenants, as defined by their respective Service Level Agreements (SLAs). This capability allows for the concurrent operation of various services with distinct Quality of Service (QoS) requirements. Efficient resource allocation among slices and users with different SLAs and QoS requirements is a critical challenge in this context. The increasing complexity of the problem setup, due to the diversity of services, traffic, SLAs, and network algorithms, makes resource allocation a daunting task for traditional model-based methods.

Traditional model-based methods have proven increasingly inadequate for addressing this complexity. The wide variability in service types and the dynamic nature of network conditions make it difficult for these methods to scale and adapt in real-time. As a result, there has been a significant shift toward data-driven approaches, particularly those leveraging Deep Neural Networks (DNNs). These methods have the potential to learn and adapt to the intricate patterns of network behavior, promising more effective resource allocation strategies. Nonetheless, the application of DNNs in wireless resource allocation is filled with challenges. Unlike typical DNN applications, such as image classification, where processing can afford some latency, the requirements for 5G networks

are much stricter. Resource allocation decisions often need to be made within milliseconds, particularly for tasks like resource block allocation per OFDM frame. Furthermore, the cost of transmitting raw data across the network for centralized processing by a DNN can be prohibitive, both in terms of latency and bandwidth consumption. This necessitates a careful balance between the computational power of DNNs and the practical limitations of network infrastructure, especially in edge computing environments where quick decision-making is crucial.

To address these challenges, Distributed Deep Neural Network (DDNN) architectures have been proposed. These architectures distribute the layers of a DNN across different locations within the network, such as the network edge and the central cloud. This distribution allows for localized, quick decision-making at the edge, which is essential for maintaining low latency and reducing the communication overhead associated with centralized processing. If the local processing by the edge-based DNN layers yields a decision that meets the SLA requirements, the data does not need to be transmitted to the cloud, thereby saving time and resources. If further processing is needed, the intermediate data is forwarded to more sophisticated DNN layers in the cloud, where more complex decisions can be made.

The DDNN architecture employs an offloading mechanism to determine whether a decision should be made locally at the edge or if additional processing in the cloud is necessary. We implement two different offloading mechanisms: a Bayesian confidence-based approach and a data-driven module. The Bayesian approach uses dropout during inference to estimate the confidence level of local predictions, allowing the system to assess whether the decision made at the edge is likely to be accurate. If the confidence is low, the data is sent to the cloud for further processing. The data-driven module is trained on past decisions to classify data samples as either "remote" or "local", and during inference, it labels new data as either suitable for local resolution or requiring cloud-based processing.

In this thesis, we explore the potential of distributed Deep Neural Network (DNN) architectures, specifically focusing on Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, to address the complex challenges of resource allocation in 5G networks. Our investigation encompasses two key areas: (i) the offline joint training of Distributed DNNs (DDNNs), where both local and remote exits are trained together to optimize decision-making across the network, and (ii) the development of optimized online offloading mechanisms. To evaluate the effectiveness of our approach, we conducted experiments using the publicly available Milano dataset. Our results are promising: the proposed DDNN architectures were able to resolve nearly 30-40% of resource allocation decisions at the edge of the network without incurring additional SLA penalties. This performance is comparable to, and in some cases exceeds, that of state-of-the-art centralized models. By reducing reliance on centralized processing, our architecture not only enhances the efficiency of resource allocation but also mitigates the latency and bandwidth costs typically associated with transmitting large volumes of data to a central location.

Abrégé

La segmentation de réseau (network slicing) a émergé comme un concept transformateur dans l'évolution des réseaux 5G, marquant un changement significatif dans la manière dont les ressources réseau sont gérées et optimisées. Elle permet aux opérateurs de réseau de partitionner leur infrastructure physique, depuis la périphérie du réseau jusqu'au centre de données, en plusieurs tranches virtuelles. Chaque tranche peut être adaptée pour répondre aux exigences spécifiques de différents locataires, telles que définies par leurs accords de niveau de service (SLA). Cette capacité permet l'exploitation simultanée de divers services avec des exigences distinctes en matière de qualité de service (QoS). L'allocation efficace des ressources entre les tranches et les utilisateurs avec des SLA et des exigences de QoS différentes constitue un défi critique dans ce contexte. La complexité croissante de la configuration du problème, due à la diversité des services, du trafic, des SLA et des algorithmes réseau, rend l'allocation des ressources une tâche ardue pour les méthodes traditionnelles basées sur des modèles.

Les méthodes traditionnelles basées sur des modèles se révèlent de plus en plus inadéquates pour répondre à cette complexité. La grande variabilité des types de services et la nature dynamique des conditions du réseau rendent difficile la mise à l'échelle et l'adaptation en temps réel de ces méthodes. En conséquence, un changement significatif s'est opéré vers des approches basées sur les données, en particulier celles exploitant les réseaux de neurones profonds (DNN). Ces méthodes ont le potentiel d'apprendre et de s'adapter aux motifs complexes du comportement du réseau, promettant des stratégies

d'allocation des ressources plus efficaces. Néanmoins, l'application des DNNs à l'allocation des ressources sans fil est semée d'embûches. Contrairement aux applications DNN typiques, telles que la classification d'images, où le traitement peut tolérer un certain délai, les exigences pour les réseaux 5G sont beaucoup plus strictes. Les décisions d'allocation des ressources doivent souvent être prises en quelques millisecondes, notamment pour des tâches comme l'allocation de blocs de ressources par trame OFDM. De plus, le coût de transmission des données brutes à travers le réseau pour un traitement centralisé par un DNN peut être prohibitif, tant en termes de latence que de consommation de bande passante. Cela nécessite un équilibre soigneux entre la puissance de calcul des DNN et les limitations pratiques de l'infrastructure réseau, en particulier dans les environnements de calcul en périphérie (edge computing) où la prise de décision rapide est cruciale.

Pour relever ces défis, des architectures de réseaux de neurones profonds distribués (DDNN) ont été proposées. Ces architectures répartissent les couches d'un réseau de neurones profond à travers différents emplacements au sein du réseau, tels que la périphérie du réseau et le cloud central. Cette distribution permet une prise de décision localisée et rapide à la périphérie, ce qui est essentiel pour maintenir une faible latence et réduire les frais de communication associés au traitement centralisé. Si le traitement local effectué par les couches DNN basées à la périphérie permet de prendre une décision conforme aux exigences des SLA, les données n'ont pas besoin d'être transmises au cloud, ce qui permet d'économiser du temps et des ressources. Si un traitement supplémentaire est nécessaire, les données intermédiaires sont transmises à des couches DNN plus sophistiquées dans le cloud, où des décisions plus complexes peuvent être prises.

L'architecture DDNN utilise un mécanisme de déchargement (offloading) pour déterminer si une décision doit être prise localement à la périphérie ou si un traitement supplémentaire dans le cloud est nécessaire. Nous mettons en œuvre deux mécanismes de déchargement différents : une approche basée sur la confiance bayésienne et un module basé sur les données. L'approche bayésienne utilise le dropout lors de l'inférence pour

estimer le niveau de confiance des prédictions locales, permettant au système d'évaluer si la décision prise à la périphérie est susceptible d'être précise. Si la confiance est faible, les données sont envoyées au cloud pour un traitement supplémentaire. Le module basé sur les données est entraîné sur des décisions passées pour classifier les échantillons de données comme étant "distants" ou "locaux", et lors de l'inférence, il étiquette les nouvelles données comme étant adaptées à une résolution locale ou nécessitant un traitement basé sur le cloud.

Dans cette thèse, nous explorons le potentiel des architectures distribuées de réseaux de neurones profonds (DNN), en nous concentrant spécifiquement sur les réseaux de neurones convolutifs (CNN) et les réseaux de mémoire à long terme (LSTM), afin de relever les défis complexes de l'allocation des ressources dans les réseaux 5G. Notre investigation couvre deux domaines clés : (i) l'entraînement conjoint hors ligne des DNN distribués (DDNN), où les sorties locales et distantes sont entraînées ensemble pour optimiser la prise de décision à travers le réseau, et (ii) le développement de mécanismes de déchargement en ligne optimisés. Pour évaluer l'efficacité de notre approche, nous avons mené des expériences en utilisant le jeu de données public de Milan. Nos résultats sont prometteurs : les architectures DDNN proposées ont pu résoudre près de 30 à 40% des décisions d'allocation de ressources à la périphérie du réseau sans entraîner de pénalités supplémentaires liées aux SLA. Cette performance est comparable, voire supérieure dans certains cas, à celle des modèles centralisés à la pointe de la technologie. En réduisant la dépendance au traitement centralisé, notre architecture améliore non seulement l'efficacité de l'allocation des ressources, mais elle atténue également les coûts de latence et de bande passante généralement associés à la transmission de grands volumes de données vers un emplacement central.

Contents

Abstract i
Abrégé [Français] v
Contents ix
List of Figures xi
List of Tables xiii
Acronyms xv
Notations xvii

1 Introduction **1**
1.1 Contributions 6
1.2 Related Work 11
1.3 Outline of the Thesis 15

2 Data-driven Resource Allocation **17**
2.1 Slice Resource Allocation with DNN 18
2.2 Objective Function for Slice Resource Allocation 19

3 Proposed Distributed Deep Neural Network **25**
3.1 System Model 26
3.2 DDNN with One Local Exit 28
3.2.1 Local Exit 31
3.2.2 Remote Exit 31
3.3 DDNN with multiple Local Exits 32
3.3.1 Local Exits 35
3.3.2 Remote Exit 36

4 Joint Training and online Inference **37**
4.1 Offline DDNN Joint Training 37
4.2 DDNN Online Inference 44
4.2.1 Oracle-based Offloading 45
4.2.2 Bayesian Confidence-based Offloading 49
4.2.3 Optimized offloading 52

5 Performance Evaluation **57**
5.1 Data Preparation . . . 57
5.1.1 Input . . . 57
5.1.2 Output . . . 58
5.1.3 Data preprocessing . . . 58
5.2 Performance Metrics . . . 62
5.3 Experiments . . . 65
5.3.1 Resource Allocation Trade-off . . . 65
5.3.2 SLA Violations Avoidance . . . 74
5.3.3 Latency Reduction . . . 76
5.3.4 Input Size . . . 80
5.3.5 Objective Function . . . 81

6 Future Work and Conclusions **83**
6.1 Future Work . . . 83
6.2 Conclusions . . . 84

Appendices **87**

A Chapter 3 Appendices **89**
A.1 Prediction Uncertainty . . . 89

Bibliography **91**

List of Figures

1.1 Resources are shared between slices in 5G 2

2.1 Centralized DNN Schemes 19

2.2 Under- and Over-provisioning 21

2.3 Objective function in Eq. (2.3) 22

2.4 Objective function in Eq. (2.4) 23

2.5 Objective function in Eq. (2.5) 23

3.1 Distributed DNN Schemes 29

3.2 DDNN with two local exits 34

4.1 DDNN Training Schemes (Offline phase) 40

4.2 DDNN Inference Schemes (Online phase) 46

5.1 Trade-off curves (Total cost vs Percentage of samples exited locally) for three weight pairs 67

5.2 Trade-off curves (Total loss vs Percentage of samples predicted locally) . . 69

5.3 Trade-off curves (Total loss vs Percentage of samples predicted locally) . . 71

5.4 Trade-off curve for single local exit DDNN, $(w_L, w_R) = (0.1, 0.9)$ 73

5.5 Trade-off curve for DDNN with four local exits 75

5.6 Traffic demand predictions for a base station using two weight pairs, depicted in the scenario without an offloading mechanism 76

5.7 Traffic demand predictions for a base station using two weight pairs: Data forwarded without offloading mechanism 77

5.8 Trade-off curve for the model with $(w_L, w_R) = (0.9, 0.1)$ using 25 base stations 80

5.9 Trade-off curve for the model with $(w_L, w_R) = (0.9, 0.1)$ using the objective function in Eq. (2.3) with $c = 10$ and $\epsilon = 0.1$ 81

List of Tables

5.1 Models for Performance Comparison 65

5.2 Latency Comparison (milliseconds per sample) 79

Acronyms and Abbreviations

The acronyms and abbreviations used throughout the manuscript are specified in the following. They are presented here in their singular form, and their plural forms are constructed by adding and *s*, e.g. BS (base stations) and BSs (Base stations). The meaning of an acronym is also indicated the first time that it is used.

3D	Three Dimensional
5G	Fifth Generation
6G	Sixth Generation
AI	Artificial Intelligence
BS	Base Station
BW	Bandwidth
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRAN	Cloud Radio Access Network
DL	Deep Learning
DNN	Deep Neural Network
DDNN	Distributed Deep Neural Network
FC	Fully Connected
GPU	Graphics Processing Unit
GB	GigaByte
HBM2	High Bandwidth Memory-second generation
LE	Local Exit

LSTM	Long Short Term Memory
MDS	Multi-Dimensional Scaling
MEC	Multi-access Edge Computing
ML	Machine Learning
NN	Neural Network
OFDM	Orthogonal Frequency Division Multiplexing
QoS	Quality of Service
RAM	Random Access Memory
RE	Remote Exit
ReLu	Rectified Linear Unit
RTT	Round-trip Transmission Time
SGD	Stochastic Gradient Descent
SLA	Service Level Agreements
UE	User Equipment
VNF	Virtual Network Function

Notations

The next list describes an overview on the notation used throughout this manuscript. We use boldface uppercase letters ($\mathbf{A}$) for matrices, boldface lowercase letters for vectors ($\mathbf{a}$), and regular letters for scalars (a or A). Sets are represented by calligraphic uppercase letters ($\mathcal{A}$).

$\|a\|$	Absolute value of the variable a
$\|\mathbf{a}\|$	Euclidian norm of the vector $\mathbf{a}$
K	Total number of VNFs
N	The number of past samples
$\mathbf{d}_{t,N}^k$	Past N value demands of VNF k up to time t
d_t^k	VNF k value demand at time t
$\hat{y}_t^k$	Predicted allocation value for VNF k at time t
$\hat{y}_{L,t}^k$	Local exits' predicted allocation value for VNF k at time t
$\hat{y}_{R,t}^k$	Remote exits' predicted allocation value for VNF k at time t
$\mathcal{F}(\cdot;\boldsymbol{\theta})$	Whole Neural Network with its parameters
$\mathcal{F}_L(\cdot;\boldsymbol{\theta}_L)$	Local Neural Network with its parameters
$\mathcal{F}_R(\cdot;\boldsymbol{\theta}_R)$	Remote Neural Network with its parameters
$f(y,d)$	Cost paid by operator when allocation is y and actual demand is d
$\mathbf{z}_t^k$	The intermediate features generated by local Neural Network
w_L	Weight assigned to local Neural Network loss
w_R	Weight assigned to remote Neural Network loss
C_L	Total local cost, cost paid when all allocations are from local exit
C_R	Total remote cost, cost paid when all allocations are from remote exit

$\mathcal{L}$	Cost difference, $C_L - C_R$
C_T	Overall transmission cost

Chapter 1

Introduction

The advent of 5G and the ongoing evolution towards 5G+/6G networks have catalyzed significant architectural changes, prominently characterized by the integration of virtualization and resource slicing. These innovations are essential for supporting a diverse array of tenants, each with distinct Quality of Service (QoS) requirements and Service Level Agreements (SLAs). The flexibility introduced by 5G networks, particularly through the deployment of Virtual Network Functions (VNFs) at various strategic locations across the network, creates unprecedented opportunities for data-driven optimization strategies. These strategies are increasingly vital in addressing the intricate challenges associated with wireless network resource allocation Fig. 1.1.

Traditionally, the optimization of critical network components, key to maintaining service performance, has relied heavily on proprietary algorithms, heuristic methods, and simplified models. These conventional approaches often attempt to navigate the multi-objective, multi-variable optimization landscape by making extensive modeling assumptions, such as the presumed knowledge of essential input parameters and stationarity conditions. However, these assumptions frequently lead to a significant decline in the efficacy of traditional methods, especially as the dynamic and complex nature of modern networks becomes more pronounced. Moreover, the emergence of advanced tech-

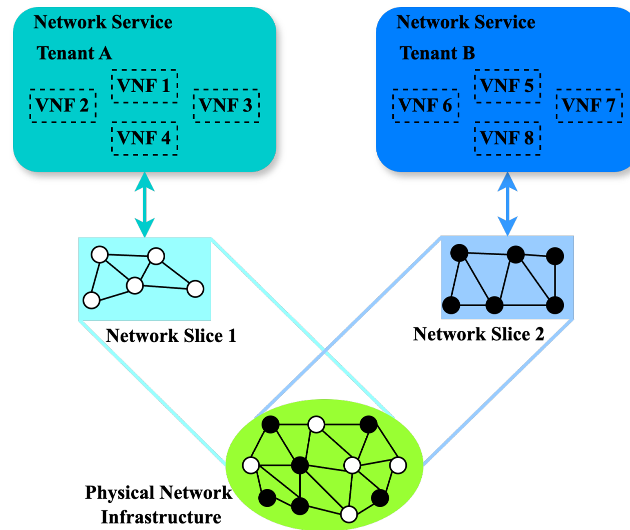


Figure 1.1: Resources are shared between slices in 5G

nologies within 5G networks has further complicated the application of these traditional, model-based optimization methods.

In response to these challenges, there has been a marked shift towards adopting more flexible, model-free algorithms derived from Machine Learning (ML) techniques. This shift represents a fundamental change in how wireless network optimization is approached, particularly in the context of beyond 5G networks. Among these ML techniques, deep learning has garnered considerable attention for its potential in slice resource allocation [1, 2, 3, 4], while reinforcement learning is being explored for its applicability in resource orchestration [5, 6, 7]. These advanced methods offer a more dynamic and responsive approach to optimizing network performance and resource utilization, moving beyond the limitations of traditional models. The increased focus on data-driven methodologies underscores the growing recognition that the complexities of modern wireless networks require equally sophisticated optimization solutions, capable of adapting in real-time to the evolving demands of next-generation networks.

Slice resource allocation utilizing deep learning has emerged as a prominent research focus within the context of 5G networks, as evidenced by recent studies [6, 7, 8, 9, 10].

In this domain, the optimization objectives are intrinsically linked to the Service Level Agreements (SLAs) that govern the provision of network resources. These objectives often display an asymmetric character, primarily due to the differing costs associated with under-provisioning and over-provisioning of resources. Under-provisioning can lead to severe consequences, including costly SLA violations and penalties paid to tenants, while over-provisioning results in inefficient resource utilization, representing a significant opportunity cost.

Addressing this asymmetry is crucial for developing effective deep learning-based solutions for resource allocation. A promising approach involves training Deep Neural Networks (DNNs) using objective functions that explicitly account for the distinct implications of under- and over-provisioning. In particular, the research presented in [11] identifies two pivotal strategies to optimize resource allocation within this framework: (i) the development of predictive models designed to achieve an optimal balance between the penalties associated with over- and under-provisioning for each network slice, and (ii) the exploitation of inherent correlations between resource requirements and slice demands by leveraging a centralized architecture based on Convolutional Neural Networks (CNNs).

In essence, the challenge lies in training DNNs to navigate the complex trade-offs inherent in network resource allocation, ensuring that the models can dynamically adjust to varying demands while minimizing the risks of both resource shortages and excesses. By incorporating tailored penalty terms into the objective function, these approaches aim to enhance the efficiency and reliability of slice resource allocation, ultimately contributing to the robustness of 5G network operations.

In this context, researchers predominantly employ centralized Deep Neural Network (DNN) architectures for network traffic prediction, leveraging historical data samples to inform resource allocation decisions [1], [4], and [11]. However, deploying such centralized, resource-intensive DNNs for resource management and control in 5G+ wireless networks introduces two critical challenges that must be addressed to ensure efficient and effective

operation.

- **Stringent Latency Requirements:** Unlike DNNs used for application layer tasks, such as image classification, which can be offloaded to centralized computational clouds with higher latency tolerance, many network optimization tasks, particularly those within the Radio Access Network (RAN), demand significantly lower latencies. For example, tasks like the allocation of RAN resource blocks among tenants or CPU allocation for Cloud Radio Access Network (CRAN) processing require near-instantaneous decision-making. Routing all relevant data to a centralized DNN for processing, followed by the transmission of the resultant control signals back to the edge components (e.g., resources for an edge Virtual Network Function (VNF)), risks violating these stringent latency requirements. The delay introduced by such centralized processing could undermine the responsiveness of the network, leading to performance degradation and potential SLA violations.

- **Overhead of Data Transmission:** Another significant challenge is the overhead associated with transmitting raw data across potentially congested edge and wireless links to a centralized, deep-core network-based DNN architecture. This data transmission not only increases latency but also consumes valuable bandwidth, which could otherwise be allocated to other critical network functions. The congestion and delays resulting from this overhead can severely impact the practicality of implementing centralized DNN solutions in real-world 5G+ environments. The excessive demand on network resources and the potential for bottlenecks necessitate a reconsideration of how and where DNNs are deployed within the network architecture.

In response to these challenges, there is growing interest in exploring the potential of distributed DNN architectures. By distributing the computational load across multiple network nodes, it may be possible to address both the latency and transmission overhead

concerns without sacrificing the performance benefits associated with DNNs. Distributed architectures offer the promise of maintaining the high accuracy and adaptability of DNN-based optimization while ensuring that decision-making processes remain within the latency and bandwidth constraints essential for 5G+ network operations. This approach represents a crucial step towards more scalable and efficient resource management in next-generation wireless networks.

Deep Neural Networks (DNNs) are conventionally structured as sequential layers, with a prediction module at the final layer. However, it is feasible and often advantageous to embed prediction modules at intermediate points within the network, thereby enabling the inference process to terminate at these “early exits” without necessitating passage through the entire network architecture. This design leverages the varying complexity of input signals, recognizing that some signals can be accurately inferred with fewer layers, thus reducing both computational overhead and latency. In such cases, predictions can be made locally at the edge, conserving resources for simpler tasks, while more complex signals that require deeper processing can be offloaded to cloud-based resources for enhanced accuracy.

Although the final exit point of the network generally yields the most accurate predictions due to the comprehensive processing involved, it also incurs significantly higher computational costs and increased latency. Therefore, identifying an “optimal” exit point within the DNN that strikes a balance between prediction accuracy and computational efficiency is critical. This optimal exit point is not fixed and can vary depending on the latent distribution of the data, the specific characteristics of the task at hand, and the operational context. The challenge lies in the fact that, during neural inference, the absence of ground truth complicates the real-time estimation of error rates at each exit point, making it difficult to select the most advantageous exit for a given inference task.

1.1 Contributions

In the context of wireless networks, particularly where edge devices with limited computational capabilities are involved, this consideration becomes even more critical. Our focus shifts to Distributed Deep Neural Networks (DDNNs), which strategically distribute the layers of a DNN across various network locations. This architecture enables early exits and local predictions at the edge when low latency or network congestion is a concern, while still allowing for remote (cloud-based) predictions when higher accuracy is required. To maximize the efficacy of this approach, it is necessary to jointly train both local and remote layers of the DDNN, ensuring that each layer, whether at the edge or in the cloud, contributes optimally to the overall performance of the network. This joint training approach not only enhances the flexibility and responsiveness of the network but also aligns with the broader goals of efficiency and accuracy in the dynamic environments characteristic of modern wireless networks.

However, most existing early exit strategies are designed with a architecture that features a single edge exit point and a central cloud, rather than incorporating multiple early exits across computational levels, which would more accurately reflect real-world scenarios. To the best of our knowledge, few studies have ventured into distributing a neural network across multiple edge computing nodes in addition to the central cloud, with the inclusion of several early exit points at the edge level. This thesis contributes to this relatively unexplored area by proposing a novel method for distributing a DNN across multiple edge devices and the central cloud, with the integration of multiple early exits. For each input sample, a critical decision must be made: whether to terminate the inference at the current exit point or to continue processing by offloading the data to the next computational level.

A notable study that addresses multi-exit scenarios at the edge is presented in [12], where the authors introduce an innovative distributed DNN architecture spanning various

computational hierarchies, including the cloud, edge, and end devices. Their approach, however, is primarily oriented towards classification tasks and utilizes an entropy-based threshold to guide offloading decisions. In such multi-exit DNN architectures, the models are often treated as black boxes with fixed thresholds, where input samples sequentially encounter exit points. At each exit point, the entropy of the output is compared to a pre-determined threshold to decide whether the inference process should terminate or continue. Higher thresholds typically trigger earlier exits, allowing for quicker, less resource-intensive predictions, while lower thresholds permit the sample to proceed further through the network layers, potentially enhancing accuracy at the cost of increased computation and latency.

The challenge lies in determining the optimal threshold for each exit point, as this threshold is heavily influenced by the underlying distribution of the input samples, a distribution that is usually unknown beforehand. This uncertainty complicates the process of setting thresholds that are both effective and efficient across different tasks and scenarios. Therefore, developing more dynamic and adaptive strategies for threshold selection, possibly informed by analysis of data, is another novelty of this research. By advancing beyond fixed-threshold approaches, it is possible to create more responsive and context-aware DNNs that can better balance the trade-offs between computational efficiency and prediction accuracy in distributed edge-cloud environments.

The primary objective of this thesis is to introduce, develop, and rigorously analyze a distributed architecture tailored to address the complex challenges of data-driven edge resource allocation. This research highlights the versatility and efficacy of our approach by implementing and evaluating two prominent Deep Neural Network (DNN) architectures: Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. By leveraging these architectures, we aim to provide a robust solution to optimize resource allocation in edge computing environments.

Training a multi-exit model within this distributed framework presents several key

challenges, including the optimal placement of exit points, the design of the exit branches, and the development of effective training strategies for these intermediate exits. Each of these factors plays a critical role in ensuring that the model can make accurate and efficient decisions at various stages of the inference process. Additionally, evaluating the performance of such multi-exit models is particularly challenging due to the need to balance computational efficiency with prediction accuracy, especially in dynamic, resource-constrained environments.

A secondary objective of this thesis is to extend our distributed architecture to address multi-edge resource allocation problems, further emphasizing the adaptability of our methodology. This extension involves the introduction and training of the architecture and also analysis of its performance across different edge computing scenarios.

Our main contributions are outlined as follows:

- (*Architecture*) We propose and train Distributed Deep Neural Network (DDNN) architectures specifically designed to address the complexities of optimizing resource allocation among distinct slices within 5G wireless networks. Our architecture is strategically structured to balance the computational demands between the network edge and the central cloud, thereby enhancing both efficiency and scalability. The architecture is composed of two key components:
  - **Shallow DNN with Local Exit:** The first component is a shallow DNN, consisting of a small number of units, positioned at the network edge. This shallow DNN is equipped with a "local exit", a prediction layer that enables rapid inferences directly at the edge. This design allows for quick decision-making, particularly for less complex tasks, thereby reducing the latency and bandwidth consumption typically associated with transmitting data to the cloud.
  - **Heavy-Duty DNN with Remote Exit:** The second component is a more extensive and computationally intensive DNN, situated within the central

cloud. This heavy-duty DNN includes a "remote exit", which processes the intermediate output from the edge network. The central cloud is responsible for handling more complex samples that require deeper inference, ensuring that the most accurate predictions are made when necessary.

Furthermore, we extend this architecture by introducing a novel distributed DNN framework that incorporates multiple separate edge exits. Each edge exit is associated with a distinct shallow DNN at the network edge, each equipped with its own "local exit" to facilitate rapid local inferences tailored to the specific demands of different slices or tasks. This distributed approach allows for even greater flexibility, as it enables the network to make localized decisions based on real-time data, while still leveraging the powerful processing capabilities of the central cloud for more demanding tasks.

The central cloud's role in this architecture is to aggregate the outputs from the various edge networks, processing them through a more comprehensive set of units to produce high-confidence predictions at its "remote exit". This design ensures that the network can efficiently manage the trade-offs between computational load, latency, and prediction accuracy, making it particularly well-suited for the dynamic and diverse requirements of 5G wireless networks.

- (*Offline Optimization*) We emphasize the critical importance of fine-tuning the joint training hyperparameters for both local and remote exits in our DDNN architecture. The primary objective of this optimization process is to achieve a delicate balance between two key goals: enabling the local layers at the network edge to make a significant number of accurate allocation decisions independently, and ensuring that these local layers generate rich, informative features that can be effectively utilized by the remote layers in the central cloud.

  This balance is crucial for the overall performance of the network. On the one

hand, the local layers need to be sufficiently empowered to handle a substantial portion of the decision-making process directly at the edge, which reduces latency and conserves bandwidth by minimizing the need to offload tasks to the cloud. On the other hand, when more complex or higher-accuracy decisions are required, the remote layers must be able to leverage the features produced by the local layers to refine and enhance these decisions.

- (*Online Optimization*) We propose an optimized offloading mechanism that intelligently learns to "hand-pick" specific samples which would gain significant benefit from additional processing in the cloud, thus justifying the associated communication and latency costs. This decision-making process occurs at the edge, *before any data is actually transmitted to the cloud.* The mechanism is trained on historical data samples, aiming to closely approximate the performance of an oracle in making these offloading decisions.

  - **Heuristic Confidence-Based mechanism:** To address the offloading decision for regression-like tasks, we introduce a heuristic rule designed to estimate the uncertainty of predictions at the local exit. This rule applies random dropout during multiple forward passes in the inference phase, creating a measure of uncertainty that guides the decision on whether to retain the decision locally or offload it to the remote cloud layers. The underlying principle is to offload only when the remote exit's additional processing is likely to sufficiently reduce SLA costs, making the extra latency and communication overhead worthwhile. This approach functions as a form of unsupervised learning, where the system assesses confidence levels to inform offloading decisions.
  - **Optimized mechanism:** In contrast to the heuristic approach, we adopt a more systematic and data-driven method for offloading decisions. This optimized mechanism leverages the known transmission costs between the edge and the cloud, framing the offloading decision as a binary classification problem. The

objective of this classifier is to determine whether engaging the remote cloud layers will result in a meaningful performance improvement that justifies the additional overhead. This classifier is trained after the DDNN has undergone its offline optimization phase, during which the local and remote costs are known. This method represents a form of supervised learning, where the classifier is explicitly trained to distinguish scenarios in which cloud-based processing offers a tangible advantage over local decision-making.

Our comparative analysis reveals that the proposed optimized offloading mechanism consistently outperforms the previously suggested heuristic approach. The optimized mechanism tries to mimic an oracle for this task, leading to better resource management and improved adherence to SLA requirements.

1.2 Related Work

The problem of resource allocation for network slicing has been approached from multiple perspectives, including Deep Neural Network (DNN)-based methods [11], [13], and stochastic control techniques [14]. In addition, network slicing has proven to be a fertile ground for the application of other data-driven methodologies, such as online convex optimization [15], [16], [17]. More recently, reinforcement learning [5], [18] has been applied to the challenge of resource orchestration, further expanding the toolkit available for addressing this complex problem.

Among the emerging approaches, the concept of Distributed Deep Neural Networks (DDNNs) with early exits [12], [19], [20], [21], [22] stands out as a promising research direction that remains largely under-explored. A critical consideration in these architectures is the amount of information that needs to be transmitted to the remote layers, as this directly impacts both latency and energy efficiency. Recent studies have proposed systematic methods for compressing the data sent towards the cloud, such as dynamically

adjusting the transmitted features based on current network conditions [23], [22], [24]. These strategies aim to optimize the trade-off between communication overhead and inference accuracy.

An innovative approach was proposed in [25], where a DDNN architecture featuring multiple early exits was analyzed. The authors assumed that the accuracy of early exits improves with the addition of more layers and formulated the offloading mechanism as an online learning problem, achieving regret guarantees. This method highlights the potential for optimizing offloading decisions in distributed DNNs, particularly in environments where balancing computational load and inference precision is crucial.

The foundational work that first sparked interest in splitting DNN layers between the edge and the cloud was conducted in [26]. Through extensive laboratory experiments, it was demonstrated that placing certain layers of a DNN at the user device while others reside in the cloud can offer substantial benefits. However, this pioneering study did not consider the inclusion of early exits for local sample resolution. Nevertheless, it provided valuable insights into the latency and energy implications of different wireless technologies (such as 3G, LTE, and WiFi) and processing units (CPU and GPU). These findings have laid the groundwork for subsequent research into optimizing distributed DNN architectures for enhanced performance in network slicing and other related applications.

Recent research on Distributed Deep Neural Networks (DDNNs) has largely concentrated on classification problems [27], [28], with the study in [12] being a pivotal contribution that introduced DDNNs within the context of image classification. The core principle of DDNNs involves the strategic distribution of DNN layers across various geographical locations, enabling decisions to be made either locally or remotely based on latency and accuracy constraints. However, applying this concept to regression-based challenges, such as resource allocation, presents significant new difficulties. In particular, it requires the development of reliable decision mechanisms to assess whether transmitting samples to the cloud for further processing is cost-effective when considering latency

and communication costs, a determination that in our work is governed by an optimized offloading mechanism.

In contrast to the classification focus in [12], [29], and [30], our research addresses a fundamentally different machine learning task: multivariate time series prediction. This shift introduces the need for substantially different and less intuitive offloading policies to handle the complexities of time series data. Our work diverges from the study in [12] in several critical ways: (i) two different DNN architectures, (ii) a confidence-based offloading mechanism that employs dropout at the local exit during inference, and (iii) an optimized offloading mechanism based on supervised learning. Finally, and more comprehensively, (iv) we establish two baselines, Random and Oracle, to provide a better understanding of the offloading policy's efficacy.

These key differences underscore the distinct challenges and innovations in our approach to DDNNs for multivariate time series prediction. By exploring these novel offloading mechanisms and comparing them against established baselines, we aim to provide a deeper understanding of how to effectively manage the trade-offs between accuracy, latency, and communication costs in the context of distributed neural networks for resource allocation.

The authors in [11] employed a centralized 3D-CNN model to predict the resources required by network slices associated with base stations (BSs) to mitigate the risks of both under- and over-provisioning. They demonstrated that regression-like problems in resource allocation can be effectively addressed by training popular DNN architectures with an objective function that carefully balances under- and over-provision terms. Building on this, recent work in [31] has explored a distributed adaptation of the architecture proposed in [11], incorporating a 3D-CNN for slice resource allocation. This adaptation introduces an uncertainty-based rule for the offloading mechanism, which allows for local inference exits, thereby enhancing the efficiency of the decision-making process.

Our work introduces several novel contributions that distinguish it from the approaches

presented in [11] and [31]:

- **Performance Benefits:** We demonstrate significant in both latency and allocation costs.

- **Optimized Offloading Mechanism:** We develop an offloading mechanism based on supervised learning, designed to optimize the decision of whether to process data locally or offload it to the cloud.

- **Establishment of Baselines:** To provide a comprehensive evaluation of our offloading policy, we establish two critical baselines: *Random Baseline:* This baseline represents a control scenario where offloading decisions are made without any informed strategy. *Oracle Baseline:* This idealized baseline assumes perfect knowledge, representing the best possible performance, and serves as a benchmark to gauge the effectiveness of our proposed mechanism.

- **Implementation of a DDNN Based on Recursive DNN and CNN Architectures:** We implement DDNNs based on both a recursive DNN architecture and a Convolutional Neural Network and showing the generality of the method.

- **Multiple Early Exits:** We implement some models that incorporate multiple early exits, representing a more practical model.

These innovations collectively contribute to a more robust and efficient approach to resource allocation in 5G networks, extending the capabilities of DDNNs beyond the frameworks previously explored in [11] and [31]. By integrating offloading mechanisms and leveraging multiple early exits, our work provides a clearer understanding of the trade-offs involved in distributed inference and offers solutions for optimizing performance in complex network environments.

1.3 Outline of the Thesis

The outline of this thesis is organized as follows:

- **Chapter 2 (Data-driven Resource Allocation):** This chapter introduces the problem setup, detailing the challenges of resource allocation in 5G networks and the motivation for employing data-driven methodologies to overcome these complexities.

- **Chapter 3 (Proposed Distributed Deep Neural Network):** In this chapter, we elaborate on the design and implementation of the proposed DDNN architectures, focusing on the integration of early exits and the specific adaptations that make the architectures suitable for efficient resource allocation.

- **Chapter 4 (Joint Training and online Inference):** This chapter explores both the offline joint training process and the online inference mechanisms. It outlines the strategies and methodologies for optimizing local and remote exits in DDNN architectures, focusing on efficient training processes. Additionally, it presents the developed and optimized offloading mechanisms that enable real-time, effective resource allocation decisions, ensuring both local and remote exits function cohesively for optimal performance.

- **Chapter 5 (Performance Evaluation):** In this chapter, the proposed architectures are validated using real-world traffic data. The chapter provides a detailed analysis of performance metrics, including latency, resource allocation costs, along with comparisons to baseline models.

- **Chapter 6 (Future Work and Conclusions):** This final chapter summarizes the main findings of the study, discusses potential future research directions, and concludes with final remarks on the implications of our work for distributed resource allocation in wireless networks.

Chapter 2

Data-driven Resource Allocation

This chapter outlines the application of centralized Deep Neural Networks (DNNs) to slice resource allocation, focusing on how these architectures can be effectively used to predict traffic demand and optimize resource allocation. In the following chapter, we will explore the feasibility of distributing and training this architecture across edge locations, assuming these locations possess the necessary computational capabilities to support such operations.

To begin, we discuss how DNNs can be effectively applied for traffic demand prediction. By leveraging historical network traffic data, DNNs are capable of learning patterns in traffic behavior, enabling accurate predictions of future demand. These predictions are crucial for optimizing resource allocation across network slices, enabling proactive management of network resources.

Following this, we discuss the selection of objective functions tailored to address the challenges of over- and under-provisioning in resource allocation. The objective function plays a key role in balancing the trade-offs between under-provisioning, which may lead to SLA violations and associated penalties, and over-provisioning, which results in inefficient use of resources. We discuss how to design an objective function that integrates both these cost factors, ensuring that the network can adaptively allocate resources in a way

that minimizes inefficiencies while maintaining service quality.

2.1 Slice Resource Allocation with DNN

We assume a network infrastructure scenario in which a provider hosts multiple network slices, each consisting of various Virtual Network Functions (VNFs). In this setup, we will associate each VNF with a discrete-time signal or time series, representing the computational resources required by these VNFs, such as CPU, memory, and bandwidth. Let the set of VNFs be denoted as $\mathcal{K} = \{1, ..., K\}$ and and the resource demand of VNF k at time t be represented by d_t^k. To effectively manage resources, we maintain a history of the previous N traffic samples for each VNF. Our objective is to leverage the history of past demands to make efficient, real-time resource allocation decisions for all $\mathcal{K}$ VNFs using a DNN-based architecture.

In the subsequent sections, we will proceed with the assumption that a DNN-based framework is employed to allocate resources across the set of $\mathcal{K}$ network functions. It is crucial to emphasize that our primary objective is not to precisely align these resource allocations with the future, as yet unknown, demand of the VNFs. Instead, the focus is on achieving a balanced trade-off between the costs associated with under-provisioning, which could lead to SLA violations, and over-provisioning, which results in resource wastage. We formulate the DNN-based resource allocation problem as follows:

$$\hat{y}_t^k = \mathcal{F}(\mathbf{d}_{t,N}^k; \boldsymbol{\theta}), \tag{2.1}$$

where $\mathbf{d}_{t,N}^k$ is the input to the DNN, defined as $\mathbf{d}_{t,N}^k = \{d_{t-N}^k, \dots, d_{t-1}^k\}$, which represents the last N traffic samples for VNF $k \in \mathcal{K}$ prior to time t. Here, N denotes the size of the input vector, which remains fixed during the model training process. The function $\mathcal{F}(\cdot; \boldsymbol{\theta})$ represents the DNN, serving as an approximation function parameterized by $\boldsymbol{\theta}$, the vector of model parameters (i.e., weights of the DNN). The output, $\hat{y}_t^k$, is the DNN's

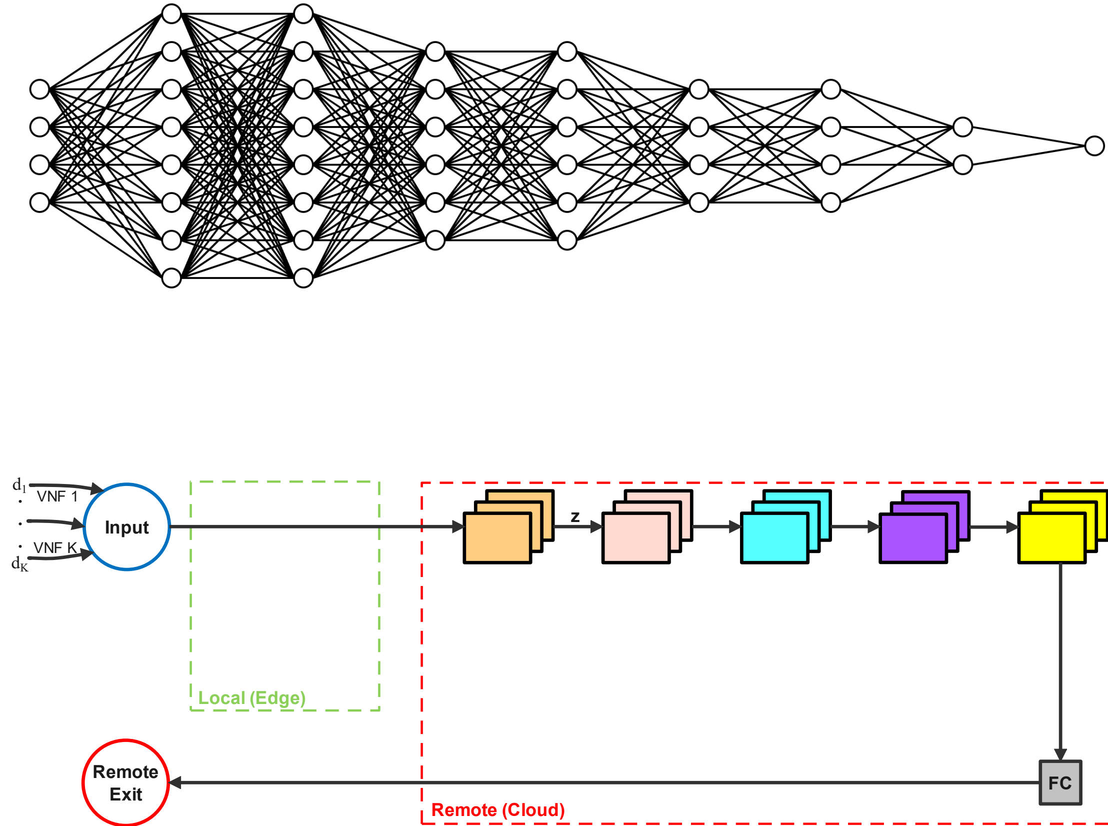


Figure 2.1: Centralized DNN Schemes

forecast for the resource demand of network element k at time t, intended to balance the trade-off between the costs of under- and over-provisioning against the expected (unknown) demand d_t^k at that time. Such a DNN network is illustrated in Fig. 2.1.

2.2 Objective Function for Slice Resource Allocation

In the realm of standard traffic forecasting, the objective is to predict traffic at a given time t based on a series of past N traffic samples, with the aim for the predicted value $\hat{y}_t$ to closely approximate the actual traffic d_t. This is traditionally achieved through training a DNN with a least squares objective function.

In standard traffic forecasting, the primary goal is to predict the traffic at a given

time t based on a series of past N traffic samples. Specifically, the objective is for the predicted value $\hat{y}_t$ to approximate the actual traffic d_t as closely as possible. This can be accomplished by training a DNN using a least squares objective function, which minimizes the difference between the predicted and actual values [32], [33].

In such a forecasting problem, the input consists of the past N traffic samples, denoted as $\mathbf{d}_{t,N} = \{d_{t-N}, \ldots, d_{t-1}\}$, and the aim is for the DNN to generate a prediction $\hat{y}_t$ that closely matches the true traffic value d_t. The training process involves optimizing the DNN's parameters by minimizing the least squares loss function, which effectively penalizes large deviations between $\hat{y}_t$ and d_t, thus improving the model's accuracy over time.

$$f(\hat{y}_t, d_t) = (\hat{y}_t - d_t)^2. \tag{2.2}$$

As previously discussed, the standard objective in traffic prediction is to forecast a value that is as close as possible to the actual value, without concern for whether the predicted value is higher or lower than the real demand. However, a key distinction in our work is that the predicted traffic demand is not an end in itself; it directly informs the allocation of resources to network elements. In this context, the direction of error, whether the predicted traffic is more or less than the actual traffic, becomes critical.

If the predicted traffic $\hat{y}_t$ is less than the actual demand d_t, this results in under-provisioning, where insufficient resources are allocated to the corresponding network element. Under-provisioning ($\hat{y}_t < d_t$) risks violating the Service Level Agreement (SLA) with the network slice tenants, leading to potential service disruptions. On the other hand, if the predicted traffic $\hat{y}_t$ exceeds the actual demand d_t, this results in over-provisioning ($\hat{y}_t > d_t$). While over-provisioning may keep the tenants satisfied, it leads to inefficiencies by allocating more resources than necessary, wasting valuable network capacity (Fig. 2.2).

To address these asymmetrical costs, our approach seeks to design an objective function that specifically minimizes the risks of under-provisioning while also reducing

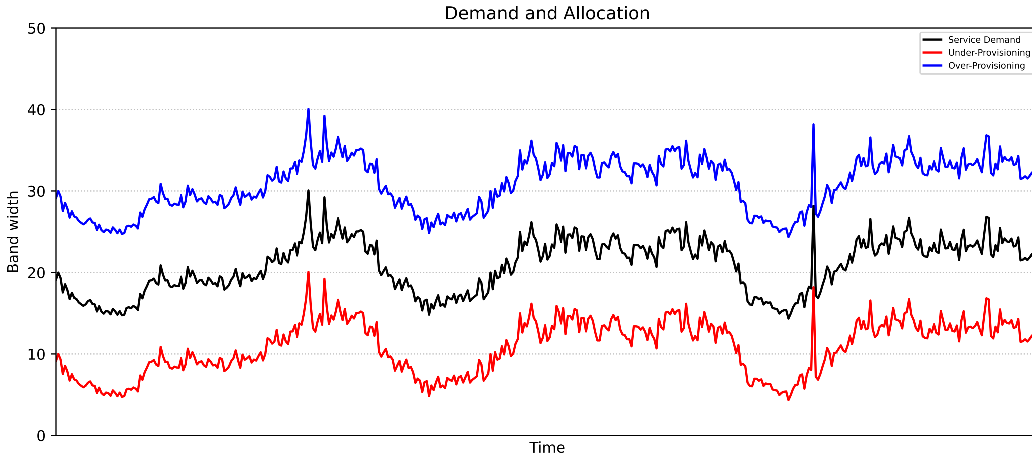


Figure 2.2: Under- and Over-provisioning

the extent of over-provisioning. Unlike conventional models that treat prediction errors symmetrically, our model tailors the DNN to prioritize avoiding SLA violations (under-provisioning), while also optimizing resource efficiency by minimizing over-provisioning.

In [11], the authors introduce the following objective function for resource allocation:

$$f(\hat{y}_t, d_t) = \begin{cases} c - \epsilon \cdot (\hat{y}_t - d_t) & \text{if } (\hat{y}_t - d_t) \leq 0 \\ c - \frac{1}{\epsilon} \cdot (\hat{y}_t - d_t) & \text{if } 0 < (\hat{y}_t - d_t) \leq \epsilon \cdot c \\ (\hat{y}_t - d_t) - \epsilon \cdot c & \text{if } (\hat{y}_t - d_t) > \epsilon \cdot c, \end{cases} \tag{2.3}$$

where ϵ is a small constant introduced to facilitate the effective operation of the Stochastic Gradient Descent (SGD) algorithm during the training of the DNN. The constant c serves as a penalty factor, designed to discourage under-provisioning. Additionally, the linear penalty in the function addresses over-provisioning, seeking to minimize the allocation of excess resources. This objective function captures the asymmetry between under-provisioning and over-provisioning costs, ensuring that the DNN adjusts resource allocation to strike an optimal balance. This function is visualized in Fig. 2.3

The authors in [31] propose the following objective function:

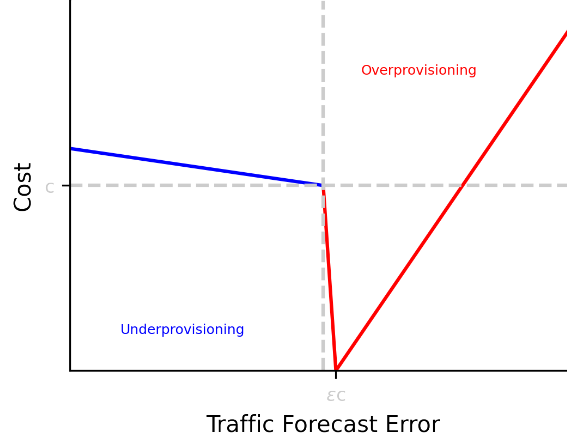


Figure 2.3: Objective function in Eq. (2.3)

$$f(\hat{y}_t, d_t) = \begin{cases} c + c_1 \cdot (\hat{y}_t - d_t)^2 & \text{if } (\hat{y}_t - d_t) \leq 0 \\ c - \frac{1}{\epsilon} \cdot (\hat{y}_t - d_t) & \text{if } 0 < (\hat{y}_t - d_t) \leq \epsilon \cdot c \\ (\hat{y}_t - d_t) - \epsilon \cdot c & \text{if } (\hat{y}_t - d_t) > \epsilon \cdot c \end{cases} \tag{2.4}$$

This function introduces a quadratic penalty, $c_1 \cdot (\hat{y}_t - d_t)^2$, for under-provisioning, thereby intensifying the penalty as the gap between predicted and actual demand increases. The quadratic term ensures that under-provisioning is penalized more severely than over-provisioning. The other terms, similar to previous formulations, address over-provisioning, with a linear penalty for moderate overestimations and a flat cost for extreme over-provisioning beyond a threshold $\epsilon \cdot c$. This approach is designed to prioritize preventing under-provisioning while maintaining resource efficiency. The structure of this objective function is visualized in Fig. 2.4, illustrating how penalties are assigned based on the difference between predicted and actual traffic demands.

Without loss of generality, we will adopt the following objective function:

$$f(\hat{y}_t, d_t) = \begin{cases} c_1 \cdot (\hat{y}_t - d_t)^2 & \text{if } (\hat{y}_t - d_t) \leq 0 \\ c_2 \cdot (\hat{y}_t - d_t) & \text{if } (\hat{y}_t - d_t) > 0, \end{cases} \tag{2.5}$$

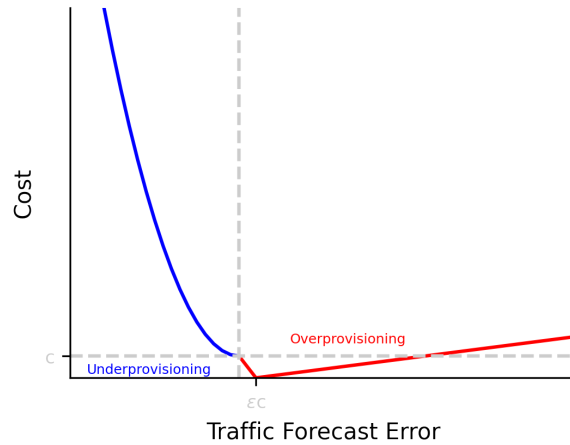


Figure 2.4: Objective function in Eq. (2.4)

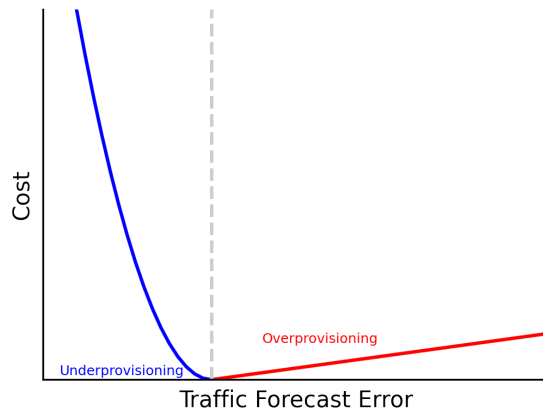


Figure 2.5: Objective function in Eq. (2.5)

where SLA violations (under-provisioning) are penalized quadratically, reflecting the severe consequences of under-provisioning, while the "opportunity cost" of over-provisioning, i.e. wasted resources, incurs a linear penalty (e.g., reflecting the money that another tenant would be willing to pay per unused resource unit). This formulation emphasizes avoiding under-provisioning by applying a heavier penalty when the predicted traffic $\hat{y}_t$ falls below the actual demand d_t, while still accounting for the inefficiencies of over-provisioning. This objective function is depicted in Fig. 2.5, illustrating the non-symmetric treatment of under- and over-provisioning costs.

It is important to note that our architecture is flexible and can accommodate various non-symmetric objective functions, similar to those presented in [11] and [31]. Our algorithms are orthogonal to the specific training objective function used, meaning they are adaptable and can function effectively regardless of the particular cost model employed.

Chapter 3

Proposed Distributed Deep Neural Network

A significant limitation of centralized DNNs lies in their inability to effectively address several critical factors that are essential in modern network environments. First, centralized DNNs introduce substantial latency due to the round-trip communication between the edge devices, where data is generated, and the cloud, where computations are performed. This latency can negatively impact real-time decision-making, which is especially crucial in latency-sensitive applications.

Second, the large volume of data that must be transmitted from the edge to the cloud can strain network bandwidth and lead to increased communication costs. In scenarios where massive amounts of real-time traffic data need to be processed this data transmission can create bottlenecks and slow down overall system performance.

Finally, there are significant privacy concerns associated with sending sensitive or proprietary data to a centralized cloud for processing. When data is transmitted over potentially insecure networks, the risk of data breaches or unauthorized access increases, raising concerns about data protection.

To address these challenges, we propose a distributed DNN architecture that strategi-

cally partitions the computational workload between the edge and the cloud. By enabling localized processing at the edge, the system can handle certain tasks immediately without requiring a full round-trip to the cloud, thereby reducing latency and improving the responsiveness of the system. Moreover, the reduction in data transmission lowers both bandwidth usage and communication costs, making the system more efficient and scalable for handling large volumes of data.

Additionally, by keeping sensitive data at the edge and limiting the amount of information sent to centralized servers, this approach enhances privacy protection. By processing data locally when possible, the system mitigates the risks associated with transferring sensitive information across the network, thereby addressing privacy concerns and improving data security.

In summary, this distributed DNN approach not only improves system efficiency and reduces latency but also offers a more privacy-conscious solution, making it particularly well-suited for modern, dynamic network environments that demand real-time, secure, and efficient decision-making.

3.1 System Model

To overcome the challenges and limitations associated with centralized DNNs, we have adopted a distributed DNN approach, which leverages the computational capabilities of edge computing. We assume that while edge devices can handle a portion of the DNN's computational workload, they may not have the capacity to process the entire model. Nevertheless, this setup enables the edge to make immediate and valid resource allocation decisions using its early exit feature. This architectural approach necessitates the partitioning of the DNN across two geographically distinct locations: the edge, where preliminary processing occurs, and the remote cloud, which handles more complex computations when necessary.

The core principle behind this distributed framework is based on the trade-off between

latency, communication costs, and decision accuracy. For certain network traffic samples, the allocation of resources via the cloud-based DNN layers incurs a higher overall cost compared to making an immediate allocation decision at the edge. This cost differential stems from the extra expenses that the network operator incurs when transmitting data across the wireless channel to the cloud. Specifically, two distinct types of costs must be considered in this scenario.

First, there is the cost associated with suboptimal resource allocation, particularly in terms of under-provisioning or over-provisioning. Under-provisioning can lead to service degradation and potential SLA violations, while over-provisioning results in resource inefficiencies, as excess computational resources are allocated without corresponding demand. Both of these conditions are financially detrimental to network operators, making it critical to achieve a balance between meeting traffic demand and minimizing wasted resources.

Second, there is the cost linked to the communication overhead required for engaging with the cloud-based DNN. This communication cost includes the additional *latency* introduced by transmitting features from the edge to the cloud and waiting for the resource allocation decision to be relayed back to the edge device. This latency not only affects the timeliness of decision-making but also impacts user experience and overall network performance. Additionally, the *communication cost* itself refers to the operational expenses incurred in sending and receiving data over the network, which can scale up significantly as the volume of data increases.

In light of these factors, our approach aims to optimize the balance between using local edge resources for immediate decision-making and deferring to the cloud for more complex tasks that may require higher accuracy but come at the cost of increased latency and communication overhead. This trade-off will be examined in greater detail in the performance evaluation section, where we quantify the impact of these costs on the overall performance of the distributed DNN architecture. Through this analysis, we will

demonstrate how the distributed approach can outperform traditional centralized models in terms of both resource efficiency and latency.

3.2 DDNN with One Local Exit

In our problem formulation for a 5G network scenario, we consider a set of VNFs, each of which requires specific computational resources such as CPU, memory, and bandwidth to fulfill the SLAs of its users. At any given time t, a VNF demands a particular amount of these resources, denoted as d_t^k, where k refers to the specific VNF. To manage these demands effectively, we track the historical demand values $\mathbf{d}_{t,N}^k$, which represent the past N traffic samples for each VNF. These demand samples are often random and potentially non-stationary due to the dynamic nature of network traffic, making prediction and resource allocation challenging.

The historical demand vector $\mathbf{d}_{t,N}^k$ is then input into a DDNN, which is tasked with determining the appropriate resource allocation for each VNF at time t, represented as $\hat{y}_t^k$. Unlike traditional DNN models, the DDNN architecture is designed to handle the distributed nature of the network, with computation spread across both edge and cloud locations, thereby offering flexibility through local exits.

The operational behavior of the DDNN can be encapsulated in the following equation:

$$(\hat{y}_{L,t}^k, \hat{y}_{R,t}^k) = \mathcal{F}(\mathbf{d}_{t,N}^k; \boldsymbol{\theta}_{\text{DDNN}}), \tag{3.1}$$

where $\mathcal{F}(\cdot; \boldsymbol{\theta}_{\text{DDNN}}))$ represents the approximation function of the DDNN, and $\boldsymbol{\theta}_{\text{DDNN}}$ denotes the parameters of the model learned during the training process.

This DDNN architecture diverges from traditional models such as the one described in Eq. (2.1) by incorporating a dual-output feature. The network produces two distinct outputs: $\hat{y}_{L,t}^k$, which corresponds to the local exit, and $\hat{y}_{R,t}^k$, which corresponds to the remote exit in the cloud. This dual-output mechanism reflects the distributed computation

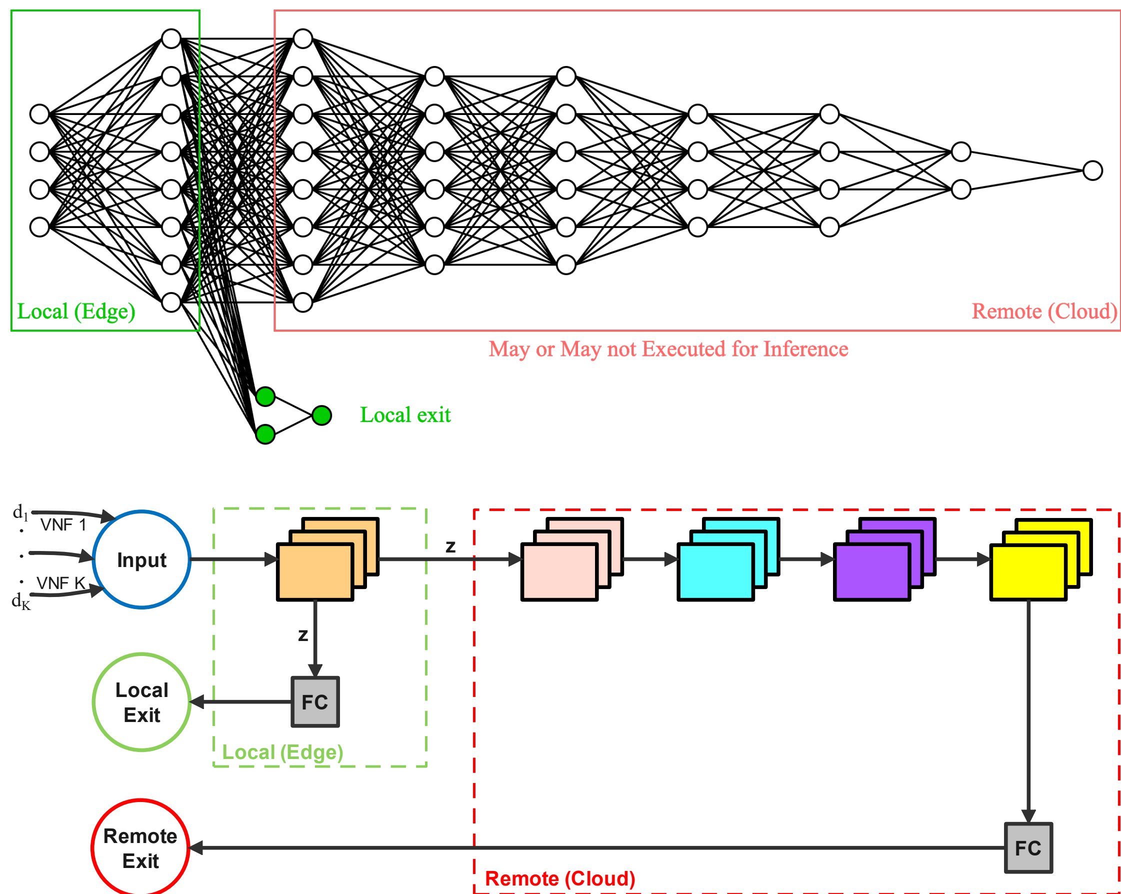


Figure 3.1: Distributed DNN Schemes

across the edge and the cloud, where immediate decisions can be made locally to minimize latency, while more complex, resource-intensive decisions can be deferred to the cloud for higher accuracy. The flexibility of this design allows the system to dynamically balance the trade-offs between resource efficiency, response time, and decision accuracy based on network conditions and demand requirements.

The DDNN architecture is depicted in Fig. 3.1, highlighting its dual-level structure that spans both edge and cloud resources. This architecture incorporates an initial, compact DNN module positioned at the network edge, designed to execute local resource allocation decisions with minimal latency. The output of this local computation is then forwarded to a more comprehensive DNN module located remotely, enabling further

refinement of the decision-making process.

The local allocation decision is formulated as follows:

$$\hat{y}^k_{L,t} = \mathcal{F}_L(\mathbf{d}^k_{t,N}; \boldsymbol{\theta}_L), \tag{3.2}$$

where $\mathcal{F}_L(\cdot; \boldsymbol{\theta}_L)$ represents the function of the local DNN layers, and $\boldsymbol{\theta}_L$ denotes the corresponding parameters. The input to this local DNN module is the historical demand vector $\mathbf{d}^k_{t,N}$, which encapsulates the traffic patterns of VNF k. After processing this input, the local DNN produces an output, denoted as $\mathbf{z}^k_t$, which represents an intermediate feature vector used for further decision-making (illustrated in Fig. 3.1).

Next, this output $\mathbf{z}^k_t$ is transmitted to a secondary, more extensive DNN module, typically located remotely from the edge, such as at a Base Station (BS) or in the cloud. This module performs additional computations to make a more informed resource allocation decision. The remote decision is expressed as:

$$\hat{y}^k_{R,t} = \mathcal{F}_R(\mathbf{z}^k_t; \boldsymbol{\theta}_R), \tag{3.3}$$

where $\mathcal{F}_R(\cdot; \boldsymbol{\theta}_R)$ represents the function of the DNN layers in the remote module, and $\boldsymbol{\theta}_R$ are the parameters specific to this remote section of the DDNN. The feature vector $\mathbf{z}^k_t$, produced by the local DNN, serves as the input to this larger, more computationally intensive module, enabling the DDNN to refine the initial allocation decision made at the edge.

This hierarchical architecture demonstrates how the DDNN processes data in stages, allowing for both fast, localized decision-making and more complex, cloud-based processing when necessary.

3.2.1 Local Exit

In a DDNN, the local exit is part of the initial layers of the neural network that are situated at the network edge. It provides an early decision-making capability, allowing the system to process data locally and rapidly when the complexity of the task does not necessitate offloading to the remote cloud.

Without loss of generality, the local component of our DDNN is designed with a single DNN block, which reflects its simplicity compared to the larger, more computationally demanding remote component. During the training phase (offline mode), the output from this block, denoted as $\mathbf{z}_t^k$, is sent both to the local Fully Connected (FC) block and to the remote layers, as illustrated in Figs. 3.1 and 4.1. This simultaneous routing allows the DDNN to optimize the balance between local and remote processing during training.

In the inference phase (online mode), the intermediate output $\mathbf{z}_t^k$ is passed through an offloading decision block, which evaluates whether the processing should continue locally within the FC block or if the data should be transmitted to the more complex remote layers for further processing, as shown in Fig. 4.1.

The result of processing by the local FC block, known as the *local prediction* or *local exit inference*, is represented by $\hat{y}_{L,t}^k$. This local prediction enables the DDNN to make rapid resource allocation decisions at the edge without incurring the additional delay and communication overhead associated with sending data to the remote cloud. Thus, the local exit feature is instrumental in enhancing the overall efficiency and responsiveness of the DDNN, especially in scenarios where real-time decision-making is critical.

3.2.2 Remote Exit

In a DDNN, the remote exit refers to the component of the architecture that is typically located in a central cloud or on a server with significantly more computational resources than the edge. This remote exit allows the network to handle more complex processing tasks that may exceed the capabilities of the local layers.

Without loss of generality, the remote component of our DDNN consists of three DNN blocks, followed by a series of four Fully Connected (FC) blocks. The first three FC blocks are structured with 128, 64, and 32 hidden neurons, each utilizing the Rectified Linear Unit (ReLU) activation function. These layers are designed to progressively reduce the dimensionality and complexity of the input, thereby refining the features extracted from the intermediate signal $\mathbf{z}_t^k$, which originates from the local exit. The sequence concludes with a linear FC block, which integrates the processing results and produces the final output decision.

The intermediate signal $\mathbf{z}_t^k$, produced at the local exit, serves as the input for the remote processing blocks, as depicted in Figs. 3.1 and 4.1. Once processed by the remote layers, the output is termed the *remote prediction* or *remote exit inference*, denoted by $\hat{y}_{R,t}^k$. This remote prediction is often used in cases where the local processing is insufficient for making an accurate decision, or when higher precision is required.

By leveraging the greater computational capacity available in the remote layers, the DDNN can handle more intricate computations and optimize resource allocation decisions that require deeper analysis. The remote exit, therefore, complements the local exit by providing a higher level of decision refinement, albeit with additional latency and communication overhead. This hierarchical processing structure, combining both local and remote exits, enables the DDNN to achieve a balance between real-time responsiveness and computational accuracy.

3.3 DDNN with multiple Local Exits

The DDNN with multiple local exits extends the single local exit architecture by incorporating two or more parallel local components, each responsible for processing different portions of the data. This parallelization allows for faster and more distributed decision-making. By handling subsets of the data simultaneously, the system can make more efficient resource allocation decisions, particularly in large-scale or high-traffic

environments.

To the best of our knowledge, the utilization of local components in this parallel manner, where multiple local exits process different segments of the data independently during inference, has not been previously explored. For example, two local components might each handle half of the data, or four exits might each process a quarter. While prior research on model distribution for inference has focused on distributing exits along a computational hierarchy, with consecutive exits positioned between layers of computation [34], the parallel distribution of local exits introduces a novel approach. Additionally, much of the previous work has focused on distributing the training process, which is outside the scope of this research.

The architecture of a DDNN with multiple local exits can be described by the following equation:

$$(\hat{y}^k_{L1,t}, \hat{y}^k_{L2,t}, \hat{y}^k_{R,t}) = \mathcal{F}(\mathbf{d}^k_{t,N}; \boldsymbol{\theta}_{\text{DDNN}}), \tag{3.4}$$

where $\mathcal{F}(\cdot; \boldsymbol{\theta}_{\text{DDNN}})$ is the approximation function modeling the DDNN, and $\boldsymbol{\theta}_{\text{DDNN}}$ denotes the model parameters. The DDNN with two local exits generates three outputs: $\hat{y}^k_{L1,t}$ and $\hat{y}^k_{L2,t}$ for the two local exits, and $\hat{y}^k_{R,t}$ for the remote exit.

It is important to note that the number of local exits can be more than two, depending on the architecture's requirements. These local exits operate in parallel rather than sequentially, meaning each local component processes its respective data simultaneously. This is fundamentally different from architectures where multiple local exits are distributed sequentially across different layers of the network. By employing parallel local exits, the architecture can further reduce latency and improve responsiveness.

The architecture, as depicted in Fig. 3.2, consists of two parallel, compact DNN modules, each situated at distinct edge locations. These modules are responsible for executing partial local resource allocation decisions. Each local component processes a portion of the input data, thereby distributing the computational load between multiple

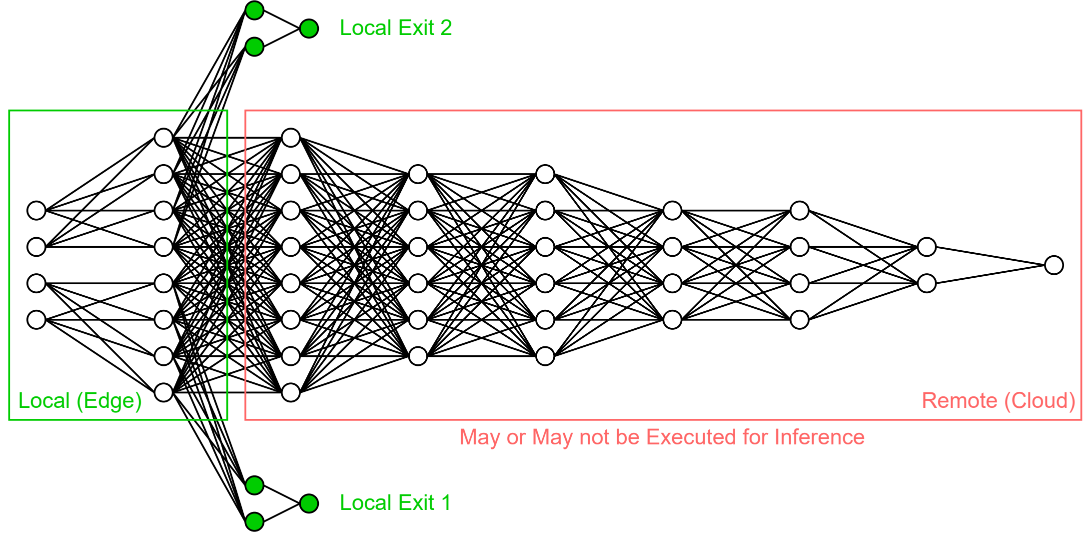


Figure 3.2: DDNN with two local exits

edge nodes.

The formulation for the local allocation decisions made by the two parallel DNN modules can be described as follows:

$$\hat{y}^k_{L1,t} = \mathcal{F}_{L1}(\mathbf{d}^k_{L1,t,N}; \boldsymbol{\theta}_{L1}), \tag{3.5}$$

$$\hat{y}^k_{L2,t} = \mathcal{F}_{L2}(\mathbf{d}^k_{L2,t,N}; \boldsymbol{\theta}_{L2}), \tag{3.6}$$

where $\mathbf{d}^k_{L1,t,N}$ and $\mathbf{d}^k_{L2,t,N}$ are the respective input data for each local DNN module, such that $\mathbf{d}^k_{L1,t,N} \cup \mathbf{d}^k_{L2,t,N} = \mathbf{d}^k_{t,N}$ and $\mathbf{d}^k_{L1,t,N} \cap \mathbf{d}^k_{L2,t,N} = \emptyset$. In other words, the total input data $\mathbf{d}^k_{t,N}$ is split between the two local DNN modules, with no overlap between the portions they handle.

Here, $\mathcal{F}_{L1}(\cdot; \boldsymbol{\theta}_{L1})$ and $\mathcal{F}_{L2}(\cdot; \boldsymbol{\theta}_{L2})$ represent the local DNNs, each with their own parameter sets $\boldsymbol{\theta}_{L1}$ and $\boldsymbol{\theta}_{L2}$. These local DNN modules aim to make quick, efficient resource allocation decisions based on their respective inputs. However, for some samples, the local decisions may not be sufficient or appropriate, potentially leading to high costs

due to suboptimal resource allocation.

In cases where the local decisions are deemed inadequate, the outputs of the two local DNNs, denoted as $\mathbf{z}_t^k = \mathbf{z}_{L1,t}^k \cup \mathbf{z}_{L2,t}^k$ (as illustrated in Figs. 3.2 and 4.2), are transmitted to a larger, more powerful DNN module located remotely, for example, at the base station or a central cloud. This remote module performs further processing to refine the resource allocation decision.

The remote module processes the combined output from the local DNN blocks and generates its own resource allocation decision, as formulated in the following equation:

$$\hat{y}_{R,t}^k = \mathcal{F}_R(\mathbf{z}_t^k; \boldsymbol{\theta}_R), \tag{3.7}$$

where the input $\mathbf{z}_t^k$, derived from the outputs of the local DNN blocks, serves as the input to the remote DNN module. The function $\mathcal{F}_R(\cdot; \boldsymbol{\theta}_R)$ represents the remote DNN layers, and $\boldsymbol{\theta}_R$ denotes the parameters specific to the remote DNN component.

3.3.1 Local Exits

In our DDNN, the local exits are part of the initial layers situated at the network edge. Without loss of generality, we assume two distinct local components, which could be geographically distributed, each containing a single DNN block. This arrangement highlights their simplicity in comparison to the more complex remote component, which handles more computationally intensive tasks.

During the training phase (offline mode), the outputs from these local blocks, denoted as $\mathbf{z}_{L1,t}^k$ and $\mathbf{z}_{L2,t}^k$, are processed by their respective local Fully Connected (FC) blocks. These outputs are then aggregated to form the signal $\mathbf{z}_t^k$, which is subsequently transmitted to the remote layers for further processing, as depicted in Fig. 4.2. The combination of these local outputs, $\hat{y}_{L,t}^k = \hat{y}_{L1,t}^k \cup \hat{y}_{L2,t}^k$, constitutes what is referred to as the *local prediction* or *local exit inference*.

This local exit inference enables the system to make swift and efficient decisions at

the network edge. In cases where the local decision-making process suffices, this can result in reduced latency and lower communication costs, as there is no need to rely on the remote layers for additional processing.

3.3.2 Remote Exit

In our DDNN, the remote exit refers to the portion of the network architecture that is typically situated in a central cloud or on a more computationally powerful server. This remote component is designed to handle tasks that require deeper processing or more complex computations than those managed by the local exits at the edge.

Without loss of generality, the remote component of our DDNN comprises three DNN blocks followed by four Fully Connected (FC) blocks. The first three FC blocks are configured with 128, 64, and 32 hidden neurons, respectively, each employing the Rectified Linear Unit (ReLU) activation function. These layers progressively refine the features extracted from the input data, reducing the dimensionality while maintaining the critical features needed for accurate decision-making. The architecture ends with a linear fully connected (FC) block, designed to consolidate the processed data and generate the final decision output.

The input for the remote blocks, $\mathbf{z}_t^k$, originates from the outputs of the local modules, as depicted in Fig. 4.2. Once processed by the remote layers, the resulting output is termed the *remote prediction* or *remote exit inference*, denoted by $\hat{y}_{R,t}^k$. This remote exit enables the network to make more accurate decisions, especially in scenarios where the local exits may not provide sufficient computational depth or where higher precision is required.

Chapter 4

Joint Training and online Inference

4.1 Offline DDNN Joint Training

Training a Distributed Deep Neural Network (DDNN) involves greater complexity compared to training a centralized DNN, as it requires the simultaneous optimization of both local and remote DNN modules under a unified objective. This process, referred to as joint training, is critical to ensure that the distributed components work cohesively to achieve efficient and accurate predictions.

The key challenge in joint training lies in balancing the contributions of both the local and remote exits in the overall objective function. The local exits, designed for quick, low-latency decision-making, must be trained to provide sufficiently accurate predictions with limited computational resources. Conversely, the remote exit, equipped with greater processing power, focuses on generating decisions with higher precision. Both components need to be aligned so that the loss function is properly backpropagated through all relevant layers of each DNN module.

During joint training, the network must simultaneously optimize the parameters of the local DNNs and the remote DNN to minimize the total loss. The objective function typically integrates both the local and remote predictions, balancing the trade-off between

fast, approximate decisions at the edge and more computationally intensive, accurate decisions at the remote server.

For instance, the local exits should not only learn to make acceptable decisions but also to generate intermediate outputs that are useful for the remote module. At the same time, the remote module must be trained to refine these intermediate representations, ensuring that the final decision (whether produced locally or remotely) aligns with the overall optimization goal. This joint training process ensures that both the local and remote components are fine-tuned to complement each other, leading to an efficient and effective resource allocation system.

The concept of jointly training local and remote (or final) exits within a neural network architecture was initially introduced in works such as [35], [36] (GoogleNet), and [19] (BranchyNet). In these earlier architectures, the primary role of local exits was to function as an additional regularization technique, aimed at improving network performance. Local exits were not originally intended to be integral components of the inference process, but rather to support the training of deeper networks by offering auxiliary outputs at intermediate layers.

However, the exploration of distributed DNNs with early exits, as highlighted in [21], [22], and [24], introduces a novel and promising research direction. This approach leverages local exits not only as auxiliary training mechanisms but also as critical parts of the inference process itself, particularly in distributed environments. By allowing parts of the network to make decisions at different stages, such as at the network edge or within local infrastructure, distributed DNNs can significantly reduce inference latency and improve resource efficiency.

This method reveals many unexplored opportunities in neural network architectures, particularly in the context of distributed computing and edge processing. It offers the potential for more adaptive and efficient models that can dynamically balance computational load between local and remote resources, optimizing both accuracy and performance in

real-time applications. This paradigm shift not only improves the scalability of neural networks but also opens new avenues for optimizing network architecture design in scenarios where low latency and computational efficiency are paramount.

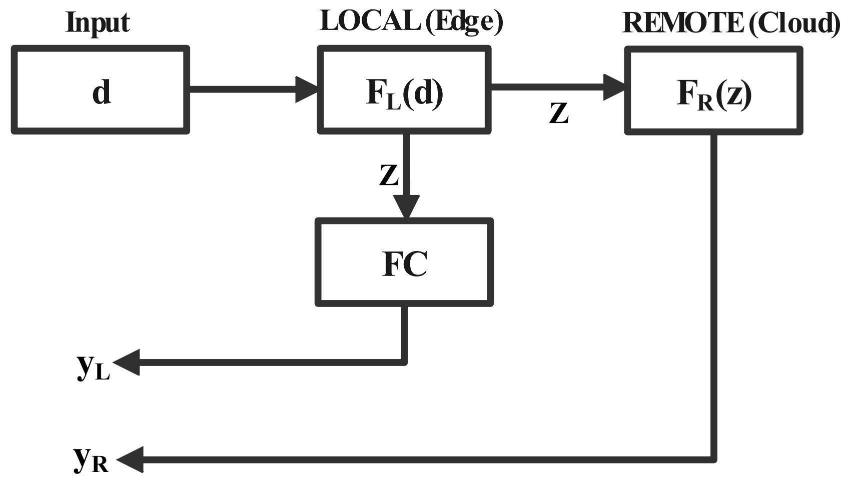
In contrast, our architecture incorporates the local exit as a core component of the inference process, making it crucial for the joint training strategy to achieve an optimal balance between the performance of both the local and remote exits. This strategy focuses on the following key performance trade-offs:

- Backpropagation for Local Exit: The performance of the local exit is backpropagated through its respective layers, denoted by $\boldsymbol{\theta}_L$). This ensures the reliability and accuracy of local decisions, represented as $\hat{y}^k_{L,t}$, even though these decisions are made by a smaller and simpler DNN module. The local exit must be effective in real-time scenarios, where low latency is critical and decisions need to be made swiftly without sending data to the remote layers.

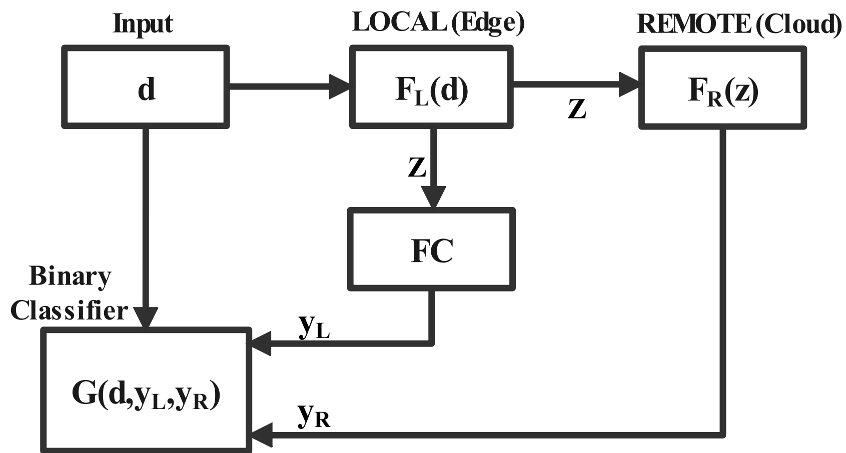
- Backpropagation for Remote Exit: The performance of the remote exit is backpropagated not only through the remote layers, denoted by $\boldsymbol{\theta}_R$, but also through the local layers, i.e., $\boldsymbol{\theta}_L$. This enhances the inference capability of the remote layers, represented by $\hat{y}^k_{R,t}$, while ensuring that the local layers generate high-quality intermediate features, $\mathbf{z}^k_t$. These intermediate features are crucial for further processing by the remote layers, facilitating a more accurate final decision.

This balanced backpropagation ensures that both the local and remote components are optimized to complement each other.

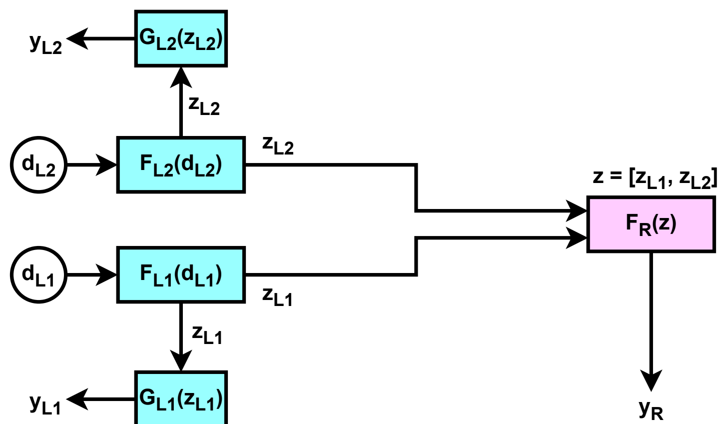
The calculation of the DDNN loss function integrates the contributions from both the edge and cloud components. At the network edge, the DDNN starts with a relatively simple DNN block, denoted as $\mathcal{F}_L(\mathbf{d}^k_{t,N}; \boldsymbol{\theta}_L)$, which handles local processing. Within the cloud infrastructure, a more complex and computationally powerful DNN block, $\mathcal{F}_R(\mathbf{z}^k_t; \boldsymbol{\theta}_R)$, performs additional refinement based on the intermediate output from the



(a) DDNN with confidence offloading mechanism



(b) DDNN with optimized offloading mechanism



(c) DDNN with two local exits

Figure 4.1: DDNN Training Schemes (Offline phase)

edge (as illustrated in Fig. 4.1).

It is important to clarify that the variation between Fig.4.1a and Fig.4.1b relates to the configuration of the offloading mechanisms and does not affect the training process of the DDNN. These configurations will be explained further in the subsequent section. The combined DDNN loss is calculated as follows:

$$
\begin{aligned}
\text{DDNN Loss} = \sum_{k=1}^{K} & w_L \cdot f(\mathcal{F}_L(\mathbf{d}_{t,N}^k; \boldsymbol{\theta}_L), d_t^k) \\
& + w_R \cdot f(\mathcal{F}_R(\mathbf{z}_t^k; \boldsymbol{\theta}_R), d_t^k) \\
= \sum_{k=1}^{K} & w_L \cdot f(\hat{y}_{L,t}^k, d_t^k) + w_R \cdot f(\hat{y}_{R,t}^k, d_t^k).
\end{aligned}
\tag{4.1}
$$

where:

- w_L and w_R represent the weighting factors that balance the contributions of the local and remote components, respectively,
- $f(\cdot, \cdot)$ is the loss function that evaluates the difference between the predicted values and the actual demand d_t^k,
- $\hat{y}_{L,t}^k$ denotes the local prediction made by the edge component, and
- $\hat{y}_{R,t}^k$ denotes the prediction made by the remote. component.

This loss function ensures that both the local and remote components are optimized during training. The local component contributes to quick, low-latency predictions, while the remote component adds higher precision to the final decision, ensuring a balance between speed and accuracy. The joint minimization of these terms allows the DDNN to efficiently allocate resources while maintaining high performance across both the edge and cloud environments.

In Eq. (4.1), the "local weight" w_L and "remote weight" w_R play a crucial role in

determining the relative impact of the local and remote exits on the overall loss during the DDNN's joint training. These weights are constrained within the range $[0, 1]$, where $w_R = 1 - w_L$, ensuring that the combined influence of the local and remote exits sums to 1. The careful selection of these weights is pivotal in the optimization process, as they control how much emphasis is placed on optimizing the local versus remote predictions.

- When $w_L = 0$ (and consequently $w_R = 1$), the DDNN effectively behaves like a centralized DNN, focusing solely on the optimization of the remote exit's performance. In this configuration, all computational efforts are geared towards the cloud or remote layers, treating local layers as auxiliary.

- Conversely, when $w_L = 1$ (and $w_R = 0$), the emphasis is shifted entirely to optimizing the local exit's performance, prioritizing localized decision-making at the edge. This setup minimizes reliance on the remote component.

The selection of the optimal w_L and w_R values depends on the desired performance outcomes. For example, a higher w_L ensures more reliable local predictions and efficient resource allocation directly at the network edge, reducing latency. On the other hand, a higher w_R leverages the remote cloud's computational power to produce more accurate but potentially slower predictions. Striking the right balance between these weights is essential for achieving efficient, low-latency decision-making, while still allowing for accurate refinements when necessary.

Our approach differs from the work in [37] (distributed learning) and [38], [39] (distributed training), where the focus is on distributing the training process of a DNN. In contrast, we concentrate on distributing the actual architecture between the edge and the core/cloud, rather than exclusively addressing the distributed training process, although such a distributed training strategy remains a feasible extension of our system.

For the DDNN with more than one local exit, the core concept remains consistent with that of the single local exit model, but the complexity increases with the introduction

of multiple local components. As an example, we present the case of a DDNN with two local exits and one remote exit. The loss computation for such a system integrates contributions from both the edge (local components) and the cloud (remote component). The formula for calculating the DDNN loss in this configuration is as follows:

$$\begin{aligned}\text{DDNN Loss} = \sum_{k=1}^{K} w_{L1} \cdot f(\mathcal{F}_{L1}(\mathbf{d}^k_{L1,t,N}; \boldsymbol{\theta}_{L1}), d^k_{L1,t}) \\ + w_{L2} \cdot f(\mathcal{F}_{L2}(\mathbf{d}^k_{L2,t,N}; \boldsymbol{\theta}_{L2}), d^k_{L2,t}) \\ + w_R \cdot f(\mathcal{F}_R(\mathbf{z}^k_t; \boldsymbol{\theta}_R), d^k_t) \\ = \sum_{k=1}^{K} w_{L1} \cdot f(\hat{y}^k_{L1,t}, d^k_{L1,t}) + w_{L2} \cdot f(\hat{y}^k_{L2,t}, d^k_{L2,t}) \\ + w_R \cdot f(\hat{y}^k_{R,t}, d^k_t).\end{aligned} \tag{4.2}$$

In Eq. (4.2), the local weights w_{L1} and w_{L2} , along with the remote weight w_R, determine the influence of each exit on the overall loss during the DDNN's joint training. These weights are critical for balancing the performance contributions from the local exits (at the network edge) and the remote exit (in the cloud).

For simplicity, and without loss of generality, we assume the local weights are equal, so that $w_L = w_{L1} = w_{L2}$. This equal weighting ensures that the two local components are treated symmetrically in the optimization process. While it is feasible to assign different weights to the local components, giving a higher weight to one would make it more influential in the training process than the other. Since the local components are parallel, setting their weights as $w_L = w_{L1} = w_{L2}$ (rather than $w_L/2$) is the correct approach.

These weights, constrained within the range $[0, 1]$, where $w_R = 1 - w_L$, play a pivotal role in the optimization process.

The optimal choice of w_L and w_R depends on the desired system behavior, such

as prioritizing local predictions for quick, low-latency decisions, or favoring remote predictions for more accurate but computationally intensive decisions. Finding the right balance between these weights is key to achieving reliable local predictions, effective resource allocation at the edge, and precise remote predictions when necessary.

4.2 DDNN Online Inference

After training the local and remote DNN modules to collaborate effectively, a crucial decision must be made during the forward pass at the local exit: *determining whether the resource allocation decision made at the local exit is sufficient, or if further refinement by the remote DNN layers is necessary.* This decision is pivotal in balancing the trade-off between low-latency decision-making at the edge and the potential benefits of more accurate, but higher-latency, processing in the cloud.

It is essential to clarify that the objective of our model is not to predict the traffic load directly. Instead, the focus is on predicting the optimal allocation of resources that minimizes the total cost associated with network operation (under-provisioning penalty and over-provisioning cost).

The local module must evaluate whether its decision for resource allocation is adequate in terms of minimizing these costs, or whether sending the intermediate features to the remote DNN for further processing will lead to a more cost-efficient outcome. This decision-making mechanism ensures that the DDNN operates efficiently, leveraging local exits for quick and approximate decisions while deferring to remote layers for more complex and critical scenarios.

Ultimately, this approach allows the system to make adaptive, cost-effective decisions in real-time, ensuring that network resources are allocated efficiently, avoiding under-provisioning (SLA violations) and minimizing over-provisioning (resource wastage), and thus optimizing both performance and resource utilization.

This decision-making process, in principle, is framed as an unsupervised learning

task. It involves selecting whether to rely on the local decision or to request additional processing from the remote layers, all while *lacking prior knowledge of the potential benefits or the extent of improvement that further processing may provide.*

The challenge lies in the fact that, at the time of the local exit, there is no explicit feedback on whether the local resource allocation is sufficient or whether the additional computational expense of sending the data to the remote layers will result in a significantly better allocation. Thus, the system must infer the decision to offload to the remote layers based on patterns in the data, model confidence, or other indicators of uncertainty that suggest further refinement could be advantageous.

The local exit must be able to assess:

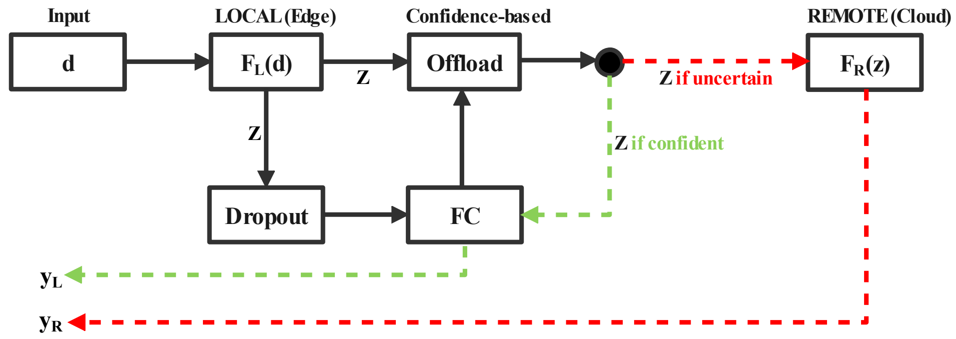
- Confidence in the Local Decision: If the local DNN has a high level of certainty about the allocation decision, further processing by the remote layers may not be necessary.

- Potential for Improvement: The system needs to estimate whether sending the intermediate results to the remote layers could result in a more optimal allocation, considering the potential cost trade-off between accuracy gains and the additional latency or communication overhead.

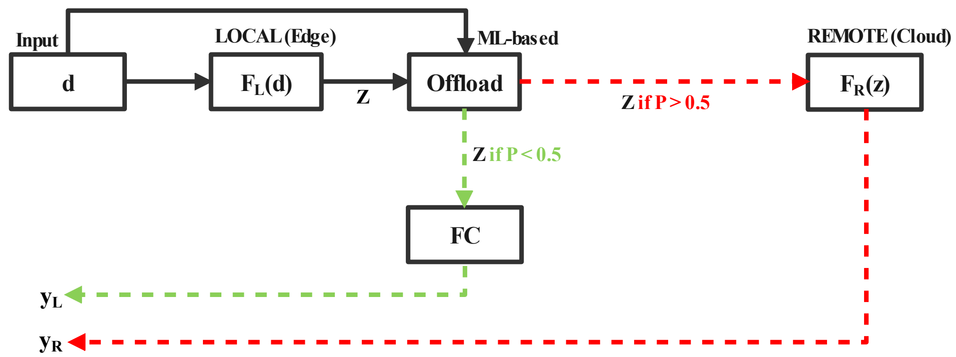
Since there is no labeled information explicitly telling the model when to stop processing at the local exit or when to seek remote refinement, the system must autonomously learn these patterns through an unsupervised task of balancing speed, resource efficiency, and accuracy. This learning approach ensures that the DDNN can dynamically adapt to different traffic conditions and network scenarios, optimizing decisions on whether to process locally or seek additional remote processing in real-time.

4.2.1 Oracle-based Offloading

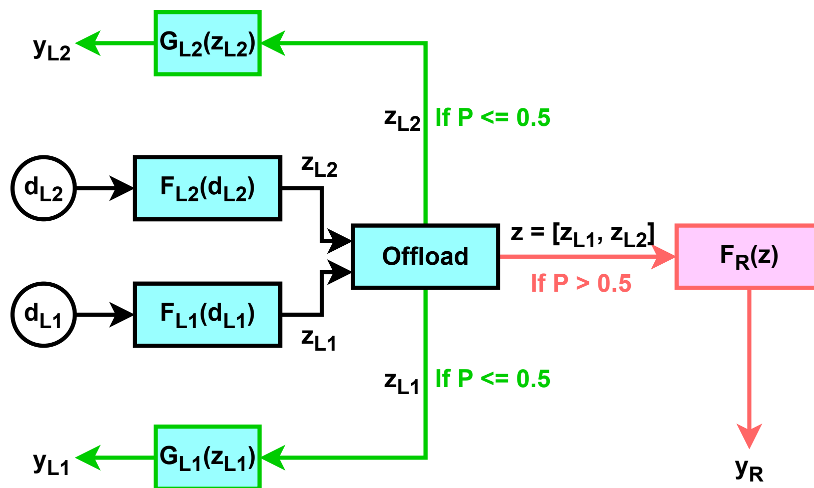
Initially, we assume the presence of an "oracle" that possesses foreknowledge of the potential improvement from remote processing for each individual sample. While such



(a) DDNN with confidence offloading mechanism



(b) DDNN with optimized offloading mechanism



(c) DDNN with two local exits

Figure 4.2: DDNN Inference Schemes (Online phase)

an oracle is unattainable in practical, real-world scenarios, its utility lies in two key areas: (a) providing a benchmark for evaluating the performance of practical offloading algorithms, and (b) demonstrating that our proposed methods can frequently approximate the theoretical lower bound established by the oracle.

The oracle is able to evaluate, for each sample, whether the remote processing will result in a significantly better resource allocation decision than the local exit. Based on this knowledge, the oracle offloads processing to the remote layers when it predicts a worthwhile improvement.

We can express the difference in costs between the local and remote decisions as follows:

$$\mathcal{L} = \sum_{k=1}^{K} (f(\hat{y}^k_{L,t}, d^k_t) - f(\hat{y}^k_{R,t}, d^k_t)) = C_L - C_R. \tag{4.3}$$

where:

- $f(\hat{y}^k_{L,t}, d^k_t)$ represents the cost of the resource allocation decision made by the local exit for VNF k,
- $f(\hat{y}^k_{R,t}, d^k_t)$ represents the cost of the decision made by the remote exit for the same VNF,
- C_L denotes the total cost associated with local decisions, and
- C_R represents the total cost associated with remote decisions.

This oracle-based approach provides an idealized benchmark, helping to measure how close our practical offloading mechanisms can come to achieving optimal decisions without foreknowledge. It serves as a theoretical ceiling for evaluating and fine-tuning our real-world algorithms.

Without loss of generality, we will also assume that there is an average cost associated with processing a sample in the cloud, denoted as C_T. This cost captures several key

factors:

- Latency Costs: This includes the additional time required to transmit the feature vector $\mathbf{z}_t^k$ from the edge to the cloud, the cloud's processing time for inference, and the relay of the resource allocation decision back to the edge.
- Communication Costs: The transmission of data to the cloud incurs potential communication costs, particularly in larger networks where significant amounts of data are sent remotely for each sample. These costs could be related to network congestion, bandwidth utilization, or even energy consumption.
- Monetary Costs: If the cloud services are provided by an external party (for example, if the DDNN has been trained and deployed by a third party as a service), there may be direct monetary costs associated with utilizing cloud infrastructure for processing.

In this thesis, we simplify the problem by assuming that this cost is a fixed and known average value, denoted as C_T, which we refer to as the **transmission cost**. This assumption allows us to focus on the decision-making process involved in determining whether to resolve a sample locally or offload it to the cloud for further processing. By comparing the trade-off between the local and remote costs, we can analyze how the transmission cost C_T affects the overall resource allocation strategy.

To determine whether a sample should be processed locally or offloaded to the cloud, we compare the cost difference $\mathcal{L}$ from Eq. (4.3) with the known transmission cost C_T. The decision-making rule is straightforward:

- If $\mathcal{L} < C_T$, this indicates that the overall cost of resolving the decision locally is lower than the combined cost of remote processing and transmission. In this case, the sample is best handled at the local exit.
- If $\mathcal{L} > C_T$, remote processing is deemed more cost-effective, and the sample should be offloaded to the cloud.

This oracle-based approach, though idealized, provides a theoretical benchmark for making optimal offloading decisions. If such an oracle existed, it would enable the system to consistently choose the most cost-effective option for each sample, balancing between local and remote processing to minimize overall costs.

An interesting complexity arises when the transmission cost C_T is neither fixed nor stationary, as might be the case in practical applications. Factors such as network congestion, variable cloud service fees, or dynamic bandwidth availability could lead to fluctuating C_T costs. In such cases, adaptive learning mechanisms, ranging from simple running averages to more sophisticated reinforcement learning techniques, could be employed to dynamically adjust to the changing environment and make better real-time offloading decisions. Exploring these adaptive approaches, though outside the scope of this paper, offers a compelling direction for future work

4.2.2 Bayesian Confidence-based Offloading

In real-world scenarios, the idealized oracle approach outlined in Eq. (4.3), which relies on knowledge of the remote decision variable, $\hat{y}^k_{R,t}$, and its associated costs, is impractical. The key challenge is that these remote predictions and their costs are unknown at the network edge without transmitting the sample to the cloud for processing. This limitation makes the oracle approach unattainable in practice.

To address this, [12] suggests leveraging the entropy of local image classifications as a measure of confidence to approximate the cost-benefit trade-off described in Eq. (4.3). In this method, high uncertainty in local predictions (as indicated by entropy) suggests that further processing in the remote layers might yield improved outcomes. However, this cross-entropy-based technique is primarily suited for classification tasks and does not directly translate to the regression-type resource allocation problems we encounter in our DDNN applications.

The literature on DDNNs, including works like [19], [34], [40], and [41], provides

limited guidance on making offloading decisions for non-classification tasks, such as regression problems. These tasks, like our focus on resource allocation, require a different approach for determining when to offload data to the remote layers.

To bridge this gap, the authors in [42] and [43] propose a Bayesian confidence-based metric, which employs random dropouts during the local forward pass to estimate uncertainty. This method offers a practical way to assess whether further processing by the remote layers is likely to improve the decision. Unlike the use of dropout as a regularization technique during training, here dropout serves as a tool for estimating model confidence in real-time. By applying dropout during the local forward pass, multiple predictions are generated, and the variance between these predictions provides an indication of the model's confidence. If the local predictions exhibit high variance (indicating low confidence), it may be beneficial to offload the sample for further refinement by the remote layers.

The confidence block consists of a dropout layer, where the dropout probability is set to $p = 0.4$, followed by a linear fully connected (FC) layer. This block is responsible for processing the intermediate signal $\mathbf{z}_t^k$, which it receives from the local DNN module. To estimate uncertainty, the confidence block performs inference on each input sample multiple times, for example, across 10 iterations (i.e., $J = 10$).

Due to the randomness introduced by the dropout process, the inference results of the confidence block vary across iterations. For each VNF in the network, denoted as $k \in \mathcal{K}$, this process generates an array of predictions in $\mathcal{R}^J$, where each element represents an inference result from one of the iterations.

To quantify the uncertainty of these predictions, the standard deviation σ_k is computed for each VNF, reflecting the variability of the model's output across the multiple iterations. This standard deviation serves as a measure of the uncertainty in the local decision for the corresponding VNF. The overall uncertainty for the network is then calculated by averaging the standard deviations across all K VNFs, yielding the final **Uncertainty** value

U:

$$U = \frac{1}{K} \sum_{k=1}^{K} \sigma_k. \tag{4.4}$$

This metric provides a worst-case estimate to quantify the potential impact of perturbations on the decisions made at the local level. The confidence mechanism evaluates the measured uncertainty value U against a predefined confidence threshold η, which is a design parameter of the DDNN. This threshold determines whether the local exit can be trusted to make an accurate decision or if further refinement by the remote layers is necessary.

- When $U < \eta$: The system interprets this as a sign that the model has high confidence in its local decision, and the decision is considered sufficiently accurate. In this case, the model proceeds with the local resource allocation (represented by the green path in Fig. 4.2a).
- When $U > \eta$: The local model exhibits significant uncertainty in its decision. In such cases, the intermediate signal $\mathbf{z}_t^k$ is forwarded to the remote layers for further processing, where a more accurate decision is expected (represented by the red path in Fig. 4.2a).

It is important to note that when deciding the resource allocation for multiple correlated elements, such as VNFs that are interdependent, the decision is made collectively, either all K decisions are processed locally, or all K decisions are forwarded to the remote layers for refinement at the same time. In future work, we plan to explore more complex hierarchies of layers and mechanisms, which could include partial views where some decisions are made locally, while others are offloaded to the remote layers. This would provide a more granular control over resource allocation and enable more efficient utilization of both local and remote computational resources.

Further theoretical and mathematical details on this method, including the probabilistic treatment of uncertainty and the impact of confidence thresholds on decision-making, can be found in Appendix A.

The confidence-based offloading mechanism, while often yielding satisfactory results (as will be demonstrated in the next section), fundamentally operates as a heuristic approach. This means it lacks formal guarantees regarding its performance or reliability, especially in complex or highly dynamic network conditions. Though effective in many cases, heuristic methods like this can lead to suboptimal decisions due to their reliance on simplified rules rather than rigorous optimization.

To address these limitations, we propose to formulate the offloading decision as a proper statistical optimization problem. By doing so, we aim to offer a more structured and verifiable approach to offloading that provides predictable outcomes. This method would move beyond heuristics by leveraging statistical models to evaluate the offloading decision, ultimately ensuring that decisions are made based on well-defined criteria.

4.2.3 Optimized offloading

We propose an offloading mechanism that leverages neural networks by implementing a "binary classifier". This classifier is responsible for determining whether a given sample should be processed locally or offloaded to the remote layers for further processing. The binary classifier is trained after the initial DDNN training process to avoid destabilizing the sensitive training path of the DDNN, particularly given the interplay between local and remote exit decisions and their corresponding weights.

The decision to train the binary classifier subsequent to the DDNN training ensures that the data flow is stable, and the classifier can effectively learn to make accurate offloading decisions without affecting the primary performance of the DDNN. This post-training step allows the offloading mechanism to assess the local and remote costs, informed by the completed DDNN training. During the training of the DDNN, all

samples pass through both the local and remote layers, providing complete data on the costs associated with each processing path. This data forms the basis for training the binary classifier, which can then make informed decisions about whether to resolve future samples locally or send them to the cloud.

The classifier uses this information to compare the transmission cost C_T and the relative costs of local and remote processing for each sample. By learning from the actual resource allocation costs observed during training, the binary classifier can accurately predict whether offloading a particular sample will result in a net benefit.

Though the classifier is currently trained independently, jointly training the offloading decision mechanism alongside the DDNN itself, where some local layers are shared between the cost minimization and offloading decision objectives, presents an intriguing area for future research. This joint training approach could further optimize resource utilization by integrating the offloading decision into the overall learning process, leading to potentially more efficient and cohesive decision-making.

In this phase of (offline) DDNN training, all samples traverse both the local and remote layers. Thus, once the DDNN is fully trained, we can calculate not only the transmission cost CT but also the relative processing costs for both local and remote exits. This comprehensive understanding of cost dynamics, gathered during training, allows for a robust and well-informed offloading decision mechanism that can optimize the balance between local processing efficiency and the benefits of remote refinement.

$$\text{Local cost: } C_L = \sum_{k=1}^{K} f(\hat{y}_{L,t}^k, d_t^k), \tag{4.5}$$

$$\text{Remote cost: } C_R = \sum_{k=1}^{K} f(\hat{y}_{R,t}^k, d_t^k). \tag{4.6}$$

The binary classifier leverages the training samples and categorizes them into two distinct classes based on the comparison of the local and remote processing costs:

- **Class 0:** $C_T < C_L - C_R$, where the remote processing cost is small enough to compensate for the transmission cost C_T. In this case, it is more efficient to offload the sample to the remote cloud for further processing.

- **Class 1:** $C_T \geq C_L - C_R$, where the remote processing cost, combined with the transmission cost, is too high, making it more advantageous to resolve the sample locally at the edge.

Without loss of generality, we employ a neural network consisting of three Fully Connected (FC) layers to serve as the binary classifier. The network outputs a probability, denoted as p, representing the likelihood of offloading a sample either locally or remotely. The classifier operates using a supervised learning approach, wherein it is trained to predict whether a sample should be processed locally or offloaded based on historical data, which includes the relative costs of local and remote processing for each sample.

This probability-based approach allows the system to make more nuanced offloading decisions, factoring in both the processing cost dynamics and the associated transmission costs. By training the classifier on past data, it learns to predict the most cost-effective resolution strategy for future samples, optimizing overall resource allocation across the network.

As illustrated in Fig. 4.2b, during inference (online mode), the input signal $\mathbf{d}_{t,N}^k$ is simultaneously fed into both the local DNN and the optimized offloading block. The offloading mechanism then computes the probability p and makes the offloading decision based on the trained classifier.

- If the classifier determines that further processing by the cloud is unnecessary (i.e., the edge-cloud round trip can be bypassed), the intermediate signal $\mathbf{z}_t^k$ is forwarded to the local fully connected (FC) layer. The resources are subsequently allocated based on the local prediction $\hat{y}_{L,t}^k$(green path in Fig. 4.2b).

- Conversely, if the classifier predicts that additional processing is required, the

intermediate signal $\mathbf{z}_t^k$ is sent to the remote DNN for further refinement. The resource allocation is then determined by the remote inference $\hat{y}_{R,t}^k$ (red path in Fig. 4.2b).

Although this approach involves an offline training phase and requires some additional hardware to implement the classifier, it is more efficient than the Bayesian confidence-based offloading mechanism. The key advantage lies in eliminating the need for J costly forward passes through the local FC during inference, which significantly reduces the computational burden at the edge.

By relying on a single forward pass through the optimized classifier, this method achieves quicker offloading decisions with minimal latency. Consequently, the system is able to maintain real-time responsiveness while optimizing resource allocation efficiency, making it particularly suited for applications where fast decision-making and low latency are critical.

The online application of the offloading block at the edge introduces a critical question: *Does the latency reduction achieved by bypassing the central cloud outweigh the additional latency introduced by the offloading block itself?* In centralized systems, the total decision-making time includes both the round-trip transmission time (RTT) to the cloud and the processing time for all samples through the entire model. This creates a uniform but potentially high-latency scenario, as every decision must traverse the cloud infrastructure.

In contrast, the Distributed Deep Neural Network (DDNN) has a different latency. The total processing time in a DDNN setup consists of several components:

- Offloading Mechanism Latency: The time taken by the offloading block to determine whether a sample should be processed locally or offloaded to the remote cloud.

- Local Processing Latency: The time required to process samples that are resolved locally at the edge, avoiding the cloud.

- Round-Trip Transmission Time (RTT): For samples that are offloaded to the cloud,

the RTT includes the time taken to send the data to the cloud and retrieve the processing results.

- Remote Processing Latency: The time taken to process the sample in the cloud.

The key benefit of the DDNN architecture is that it can reduce overall latency by resolving a substantial portion of samples locally, thus bypassing the cloud and avoiding the RTT entirely for those cases. However, this benefit comes at the cost of the additional time required for the offloading block to make its decision.

This trade-off between the latency incurred by the offloading block and the potential latency savings from avoiding cloud-based processing is crucial. The effectiveness of the offloading mechanism depends on whether it can quickly and accurately identify which samples should be processed locally versus remotely. The offloading mechanism must be efficient enough to ensure that the cumulative processing time (offloading decision + local processing) remains significantly lower than the alternative cloud-based processing path.

In the next chapter, we further explore this latency trade-off by measuring the latency. This analysis will help determine whether the introduction of the offloading block leads to net latency reduction or if the overhead negates the potential gains from avoiding cloud-based processing.

Chapter 5

Performance Evaluation

5.1 Data Preparation

To train and evaluate the proposed architecture, we employ the publicly available Milano dataset [44], which is widely used in related research studies [31], [15], and [16]. The Milano dataset provides traffic data collected from cellular Base Stations (BSs) over time, with traffic measurements recorded in megabytes (MBs). These measurements are used to simulate the traffic patterns of VNFs, offering a realistic basis for resource allocation and offloading decisions within the context of 5G network slicing.

The dataset includes time-series traffic data across different BSs in the Milano metropolitan area, which allows us to model the dynamic nature of VNF resource demands. The variability in traffic patterns across different time intervals and locations enables the architecture to learn to allocate resources efficiently under varying network conditions.

5.1.1 Input

At time t, the DDNN receives an input denoted as $(\mathbf{d}_{t,N}^{k}, K) \in \mathcal{R}^{N \times K}$. This input represents a historical snapshot of the traffic data for all $\mathcal{K}$ VNFs over a window spanning

N time intervals. Specifically:

- $\mathbf{d}_{t,N}^{k}$ corresponds to the traffic demand history for VNF k, capturing the traffic patterns observed in the preceding N time steps leading up to time t.
- The matrix $\in \mathcal{R}^{N \times K}$ consists of N rows, where each row represents the traffic data for all K VNFs at a specific time interval.

5.1.2 Output

The output of the DDNN is denoted as $\mathbf{y}_t = \{y_t^1, y_t^2, ..., y_t^K\} \in \mathcal{R}^K$, where each y_t^k represents the predicted allocation of resources for VNF $k \in \mathcal{K}$ at time t.

Specifically, y_t^k is either the local prediction $\hat{y}_{L,t}^k$ (or $\hat{y}_{L1,t}^k$ or $\hat{y}_{L2,t}^k$ for the DDNN with two local exits) or the remote prediction $\hat{y}_{R,t}^k$, depending on whether the offloading mechanism determined that the VNF's resource allocation decision should be made locally or remotely:

- Local Prediction $\hat{y}_{L,t}^k$: The resource allocation decision made at the local level (edge) without forwarding the intermediate data to the cloud.
- Remote Prediction $\hat{y}_{R,t}^k$: The resource allocation decision made after further processing by the remote cloud-based DNN layers.

5.1.3 Data preprocessing

A preprocessing step is applied to the time series data as described in [11], which is a generic procedure for both centralized and distributed configurations.

Time series data from different Base Stations (BSs) often exhibit correlations, particularly in areas near high-traffic locations such as tram lines, metro stations, and other commuter hubs. These inherent correlations between the traffic patterns of geographically proximate BSs offer a valuable opportunity to improve the efficiency and accuracy of resource allocation by deploying learning algorithms that can leverage these relationships.

By recognizing and exploiting these correlations, a shared architecture can be designed to simultaneously predict optimal resource allocations for multiple BSs in parallel. This approach enables the model to learn joint representations of traffic demand across correlated BSs.

Time series data from different BSs may exhibit correlations, for example, those near tram lines, metro stations, and other commuter hubs. These inherent correlations present an opportunity to deploy learning algorithms that can simultaneously predict optimal resource allocations for multiple BSs in parallel, utilizing the same architecture.

The preprocessing procedure organizes Base Stations (BSs) into a matrix structure where highly correlated BSs are placed adjacently, creating an "image"-like input data representation for the (D)DNN. This structured input facilitates the use of deep learning techniques that excel with spatially correlated data.

For each pair of BSs (i and j), their similarity in traffic demand, denoted as c_{ij} (referred to as correlation pairs), is computed using a time series similarity measure known as shape-based distance [45]. This distance metric captures the dynamic and temporal patterns in the traffic demand of BSs, enabling more accurate correlation analysis.

The goal is to determine the placement of the BSs in a two-dimensional space, such that their locations reflect the computed correlations c_{ij}. To achieve this, two distinct optimization problems are solved. First, let $a_k \in \mathcal{R}^2$ represent the location of BS k in a 2D plane. The optimization variable is defined as $a \in \mathcal{R}^{2\times K}$, where K is the total number of BSs. The positions of BSs, which must adhere to their correlations, are determined by solving the following optimization problem:

$$\underset{a_1,\ldots,a_K}{\text{minimize}} \sum_{i<j} (\|a_i - a_j\| - c_{ij})^2, \tag{5.1}$$

This optimization problem aims to minimize the difference between the Euclidean distance $\|a_i - a_j\|$ between BSs i and j, and their correlation c_{ij}, ensuring that BSs with

similar traffic patterns are placed closer together in the 2D space.

To solve this optimization problem efficiently, Multi-Dimensional Scaling (MDS) [46] is employed. MDS is a dimensionality reduction technique that preserves the pairwise distances between points as much as possible, making it an ideal method for placing BSs based on their correlation values. The result is a spatial arrangement of BSs that reflects their traffic demand correlations, which is then used as structured input for the deep neural network. This "image"-like input enhances the model's ability to learn from the spatial and temporal patterns present in the BS data.

To map the two-dimensional solutions obtained from the first optimization problem into a matrix format, a secondary optimization problem is introduced. The objective is to assign each BS to a point on a regular two-dimensional grid, ensuring that the spatial correlations between BSs are preserved in this grid layout.

In this step, we define a regular grid $a_1, ..., a_K$ where each point is denoted as a_i, representing the ith grid point. For each pair of i and j, a cost is computed based on the squared Euclidean distance between the grid points, expressed as:

$$e_{ij} = \|a_i - b_j\|_2^2. \tag{5.2}$$

The goal is to assign each BS to one grid point, such that the total cost is minimized. To achieve this, we introduce a binary matrix $X \in \{0, 1\}^{K_1 \times K_2}$, where each element x_{ij} indicates whether BS i is assigned to grid point j (i.e., $x_{ij} = 1$ if BS i is assigned to grid point j, and $x_{ij} = 0$ otherwise). The optimization problem is formulated as:

$$\underset{X}{\text{minimize}} \sum_{i=1}^{K_1} \sum_{j=1}^{K_2} e_{ij} x_{ij}, \tag{5.3}$$

subject to the following constraints:

$$\sum_{i=1}^{K_1} x_{ij} = 1 \quad \forall i, \quad \sum_{j=1}^{K_2} x_{ij} = 1 \quad \forall j, \tag{5.4}$$

which ensure that each BS is assigned to exactly one grid point, and each grid point hosts exactly one BS.

This optimization problem, which aims to minimize the total assignment cost while adhering to the assignment constraints, can be solved efficiently in polynomial time using the Hungarian algorithm [47].

By solving this problem, we create a structured, grid-based representation of the BSs, which forms an "image"-like input for the (D)DNN architecture, preserving the correlations between BSs and enabling more effective learning and resource allocation predictions.

LSTM-based DDNN: Long Short-Term Memory (LSTM) networks are well-suited for modeling sequential data, particularly when dealing with short time series ranging between 100 to 300 samples. For our LSTM-based DDNN, we utilize the past 144 samples, which correspond to the daily traffic measurements recorded at each Base Station (BS), to predict the next sample for a group of 16 BSs. Thus, we set the sequence length $N = 144$ and the number of VNFs $K = 16$, resulting in input data of size (K, N).

For the LSTM-based DDNN, we only solve the first optimization problem, defined in equation (5.1), to group the correlated BSs together. This ensures that the LSTM can effectively capture the temporal dependencies within each BS's traffic data while also leveraging the spatial correlations between the BSs that exhibit similar traffic patterns. By clustering the correlated BSs, the model is able to predict resource demands more accurately and efficiently, enabling simultaneous predictions for multiple base stations within the same architecture.

CNN-based DDNN: The optimal performance of 3D-CNN architectures is achieved when the input tensors exhibit a high degree of local correlation [48], [49], and [50], allowing neurons to process spatially similar values. This principle is well-illustrated in image processing, where neighboring pixels often display strong correlations. Following this approach, our goal is to develop a tensor input where adjacent elements correspond

to BSs with highly correlated mobile service demands.

To achieve this, we arrange the BSs in a structured manner, selecting $K_1 = K_2 = \sqrt{K}$ BSs[1] and applying the preprocessing steps outlined previously. Specifically, the BSs are grouped based on their correlations using the solution from the optimization problem, which ensures that neighboring grid points represent BSs with similar traffic patterns.

As a result, the input to the CNN-based DDNN is conceptualized as a "traffic box" with dimensions $\sqrt{K} \times \sqrt{K}$ over a temporal window of length N. In this setup, the grid structure enables the model to treat the traffic data similarly to how a CNN processes pixel data in an image, capturing spatial dependencies between BSs. For our experiments, we select $K = 16$, and $N = 144$, which represents the daily recorded traffic data for each BS. The input tensor fed into the CNN-based DDNN.

We set the quadratic coefficient c_1 to 50 and the linear coefficient c_2 to 1 in Eq. (2.5). The quadratic coefficient is set relatively high at 50 to emphasize the significance of deviations in the normalized traffic demand time series, which are scaled to the $[0, 1]$ range. This scaling ensures that larger deviations in the resource demands, particularly under-provisioning scenarios, incur a higher penalty, reflecting the importance of avoiding service-level agreement (SLA) violations.

For implementing our models, we use Python along with the PyTorch framework. The models are executed on the Google Colab platform, utilizing an Nvidia V100 GPU with 16 GB of HBM2 memory and 32 GB of system RAM.

5.2 Performance Metrics

We evaluate the performance of our DDNN using two primary metrics:

Percentage of Samples Resolved at the Local Edge: This metric measures the proportion of samples that are successfully processed locally, avoiding the additional latency and cost associated with sending data to the remote layers for further processing.

[1] While we use a square grid for simplicity, this approach can be adapted to more complex structures.

Overall Cost of the DDNN: The overall cost is defined as the weighted sum of the costs incurred when processing samples either locally or remotely. It is calculated as:

$$C_{\text{DDNN}} = \sum_{m=1}^{M} \mathcal{I}_m \cdot C_L^m + (1 - \mathcal{I}_m) \cdot C_R^m, \tag{5.5}$$

where:

- M is the total number of samples,
- C_L^m is the local exit cost for sample m
- C_R^m is the remote exit cost for sample m
- $\mathcal{I}_m$ is an indicator variable that determines whether a sample was processed locally or remotely, defined as:

$$\mathcal{I}_m = \begin{cases} 1 & \text{if the sample } m \text{ processed locally} \\ 0 & \text{else.} \end{cases} \tag{5.6}$$

We implement and compare the following models:

- **Centralized CNN:** This model is a fully centralized DNN, utilizing 3D-CNN algorithm as proposed in [11] for tasks similar to resource allocation. It employs a unique pre-processing method, converting time-series data into "image"-like frames for input. Both the architecture and its pre-processing methodology have been implemented for our study. Additionally, although the original DeepCog model [11] was trained using the objective function outlined in Eq. (2.3), we opted to train it using the objective function defined in Eq. (2.5). This adjustment ensures that DeepCog is directly comparable to our other models, facilitating a fair and consistent evaluation. Being entirely cloud-based, the model does not have edge prediction capabilities, which results in a local resolution percentage of zero. For quick reference, the architectures are summarized in Table 5.1.

- **Centralized LSTM:** A fully centralized LSTM architecture, indicating that all data samples are processed and resolved within the cloud. Due to its cloud-only configuration, it lacks the capability for edge predictions, resulting in a zero percentage for local resolutions. The primary purpose of utilizing this model is to highlight the inherent advantages (that we conjectured) of a recursive neural network architecture, such as LSTM, especially for handling time series data, offering a distinct advantage over the approach in [11] and [31]. It is important to note that this model is distinct from resolving all samples remotely in a DDNN framework, due to differences in their architecture and training approaches. Moreover, as noted in the foundational DDNN study [19], we will see that incorporating local exits within a DNN architecture not only differentiates the model but also introduces a unique regularization effect that influences overall performance.

- **Oracle-based DDNN:** A DDNN with the offline optimal offloading policy that operates under the premise of having complete knowledge of both exits. This model, referred to as the "Oracle", is idealistic because it assumes perfect information, which is unrealistic in real-world scenarios. The offloading decision within this model hinges on comparing the transmission cost (C_T) with the differential cost between local (C_L) and remote (C_R) exits. As C_T increases, the model tends to favor local resolutions.

- **Random-based DDNN:** A DDNN with a random offloading policy where the offloading decision for each sample is modeled as an independent and identically distributed (i.i.d.) Bernoulli random variable with a success probability p, where "success" refers to a sample being processed locally. Consequently, with a constant p, the proportion of samples processed locally effectively equals $\mathcal{L} = p$. Our simulations explore the implications of this Random policy across a range of p values within $[0, 1]$, delineating a spectrum of costs.

Table 5.1: Models for Performance Comparison

Model	Joint Training	Edge Offloading	Cloud Offloading	Realizability
Centralized LSTM	×	×	✓	✓
Centralized CNN	×	×	✓	✓
Oracle-based DDNN	✓	✓	✓	×
Random-based DDNN	✓	✓	✓	✓
Confidence-based DDNN	✓	✓	✓	✓
Optimized DDNN	✓	✓	✓	✓

- **Confidence-based DDNN:** A DDNN with the confidence-based offloading policy, as explained in the previous section. This offloading policy relies on uncertainty. The uncertainty value, U, is compared to a confidence threshold (η). Increasing η results in more samples being resolved locally.

- **Optimized DDNN:** A DDNN with the optimized offloading policy, as outlined in the previous section. The offloading mechanism functions according to the output of a binary classifier. Altering C_T changes the classifier's behavior, thereby affecting decision outputs. Increasing C_T results in a greater number of samples being resolved locally.

5.3 Experiments

5.3.1 Resource Allocation Trade-off

After training the model, we plot trade-off curves to illustrate the relationship between the total cost and the local sample resolution percentage. The model processes the test set, and the offloading mechanism determines whether each sample is handled locally or remotely, with the total cost computed using Eq. (5.5). These trade-off curves offer insights into how different offloading strategies balance the trade-off between minimizing cost and maximizing local sample resolution.

We generate the following trade-off curves:

- **Offline Oracle-based Trade-off Curve:** This curve serves as the lower bound baseline for performance. It is derived using the *Oracle-based Offloading* approach, which assumes perfect foreknowledge of the optimal offloading decision for each sample. Varying the transmission cost C_T affects the local sample resolution rate. Since the oracle makes optimal offloading decisions, this curve provides a benchmark for the best possible performance.

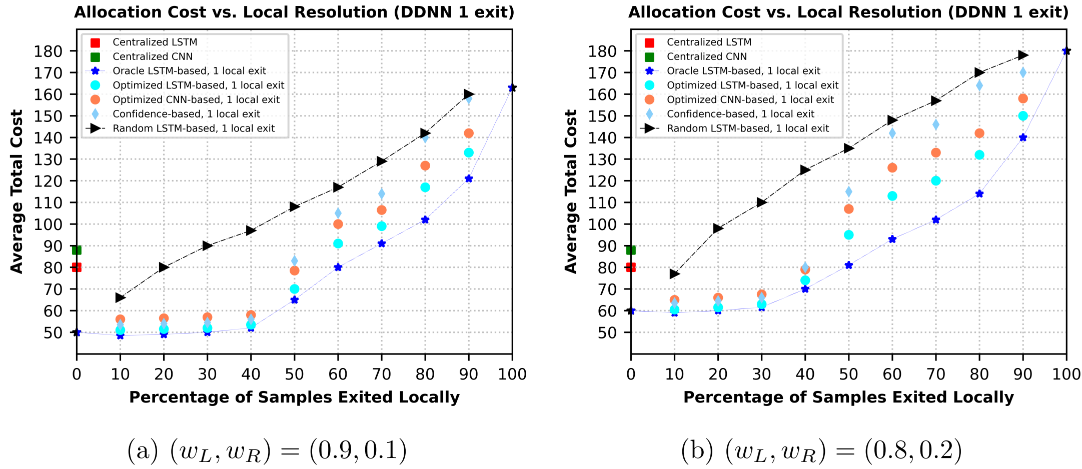
- **Random Policy-based Trade-off Curve:** This curve is used to establish an upper bound baseline for performance evaluation. It is generated by adjusting the probability parameter p from 0 to 1, which randomly determines whether each sample is processed locally or remotely. This curve shows the trade-off between cost and local resolution when decisions are made at random. Any offloading policy that results in a total cost higher than this curve is considered ineffective[2].

- **Bayesian Confidence-based Trade-off Curve:** This curve is obtained using the *Confidence-based Offloading* strategy. By adjusting the confidence threshold η within the range $[0, 1]$, the offloading mechanism selectively processes samples locally or remotely based on the uncertainty in the local decision. The curve represents the performance of this online confidence-based method.

- **Optimized Trade-off Curve:** This curve is generated using the *Optimized Offloading* approach, where increasing the transmission cost C_T influences the curve. The optimized offloading mechanism leverages the binary classifier to predict whether offloading a sample to the cloud will reduce overall cost. By adjusting C_T, the model can shift the trade-off between local processing and remote offloading, yielding a curve that reflects the cost-effectiveness of the optimized offloading decisions.

These trade-off curves provide a comprehensive visualization of the model's performance across different offloading policies, enabling a comparison of the efficiency of each

[2]Although it is technically possible to create an offloading policy that consistently selects the more costly exit, doing so would not provide a meaningful or practical benchmark for comparison.



(a) $(w_L, w_R) = (0.9, 0.1)$ (b) $(w_L, w_R) = (0.8, 0.2)$

Figure 5.1: Trade-off curves (Total cost vs Percentage of samples exited locally) for three weight pairs

method. By comparing these curves to the oracle and random baselines, we can assess how closely each strategy approximates the optimal offloading decisions.

Figs. 5.1a and 5.1b present trade-off curves for models with training weights (w_L, w_R) set to $(0.9, 0.1)$ and $(0.8, 0.2)$, respectively. A key observation from these figures is that at the operational point where no samples are exited locally (on the x-axis), the LSTM-based architecture consistently outperforms the 3D-CNN model. This advantage is particularly evident when remote exits dominate, underscoring the LSTM's superior ability to handle sequential dependencies in time-series data related to resource demands.

More importantly, the simple introduction of a local exit during training significantly boosts the baseline performance of the models, achieving improvements ranging from 20% to 50%, even though in this scenario, all samples are ultimately resolved remotely[3].

Key Observation 1: The positive impact of incorporating local exits, even within fully centralized DNNs, is clearly demonstrated in the task of optimizing resource allocation for 5G networks.

[3]The distributed model's superior performance over centralized architectures, with lower overhead, can be attributed to the influence of the local exit on gradient flow during training. This effect has been documented in other studies [12], [19], [20], showing that local exits not only enhance operational efficiency but also introduce a regularization effect, leading to improved overall performance with reduced computational and communication overhead. This creates a beneficial "win-win" situation.

The inclusion of local exits introduces several benefits:

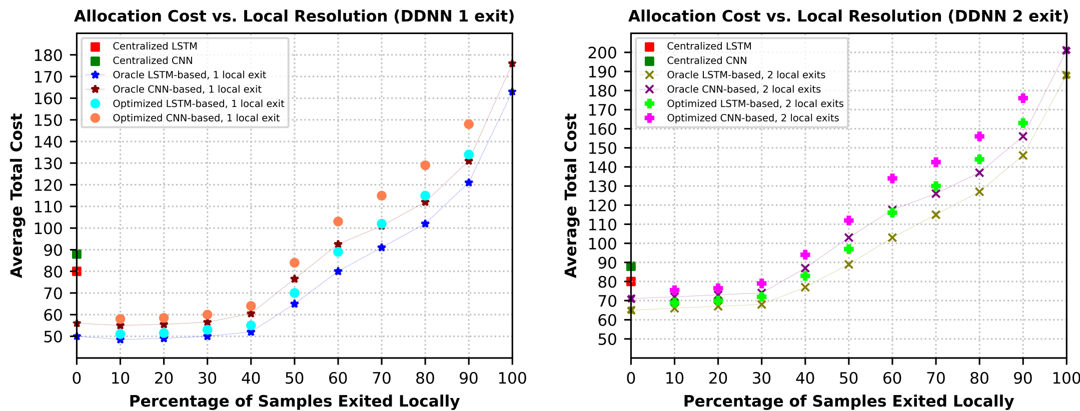
- Improved Gradient Flow: Local exits allow gradients to flow more effectively during backpropagation, reducing the likelihood of vanishing gradients in deeper networks. This enhances model convergence, leading to better overall performance.
- Operational Efficiency: By incorporating local exits, the network can resolve simpler samples earlier in the pipeline, improving operational efficiency by minimizing unnecessary computations at remote layers.
- Regularization Effect: The local exit acts as a form of regularization, encouraging the model to learn more generalizable features in earlier layers. This effect reduces overfitting and leads to a performance boost even when all decisions are ultimately made remotely.

Figs. 5.2 presents the trade-off curves for DDNN models incorporating LSTM and CNN architectures. The models with a single local exit are trained using weights set to $(w_L, w_R) = (0.9, 0.1)$, while those with two local exits are trained with weights $(w_{L1}, w_{L2}, w_R) = (0.9, 0.9, 0.1)$.

In Fig. 5.2b, the models with two local exits still improve performance by nearly 20%. The difference between models with one and two local exits lies in the fact that in the two-exit model, each local component processes half of the data, leading to lower-quality intermediate features ($\mathbf{z}_t^k$) for the remote part.

Interestingly, the model with four local exits, as depicted in Fig. 5.5, does not demonstrate any significant performance improvement over the centralized models. This suggests that further increasing the number of parallel local components leads to a diminishing return, as each local exit processes an even smaller subset of the data, resulting in progressively lower-quality intermediate features for remote processing.

Key observation 2: The incorporation of local exits has a clearly positive impact on model performance. However, increasing the number of parallel local components beyond



(a) DDNN with 1 local exit $(w_L, w_R) = (0.9, 0.1)$ (b) DDNN with 2 local exits $(w_{L1}, w_{L2}, w_R) = (0.9, 0.9, 0.1)$

Figure 5.2: Trade-off curves (Total loss vs Percentage of samples predicted locally)

a certain point can diminish the overall performance of the DDNN.

The comparison between the Oracle and Random offloading policies in both cases yields a crucial insight: certain offline offloading strategies are able to outperform the Random policy. This suggests that even without perfect information (as assumed in the Oracle), offloading strategies can be devised that are far more effective than random decision-making. This finding highlights the potential for developing online offloading policies that closely approximate the Oracle's performance, indicating that there is considerable room for improvement in offloading efficiency.

The key takeaway is that the performance gap between the Oracle and Random policies underscores the existence of learnable patterns in the data that an optimized online policy could exploit. If the Oracle, which represents the theoretical optimal performance, performs substantially better than the Random policy, it indicates that intelligent, data-driven decisions about offloading can lead to significant gains in efficiency and cost reduction. Conversely, if the performance of the Random and Oracle policies were identical, it would imply that no discernible patterns exist to inform the offloading decisions. In such a scenario, the opportunity for any online offloading policy to offer improvements would be minimal.

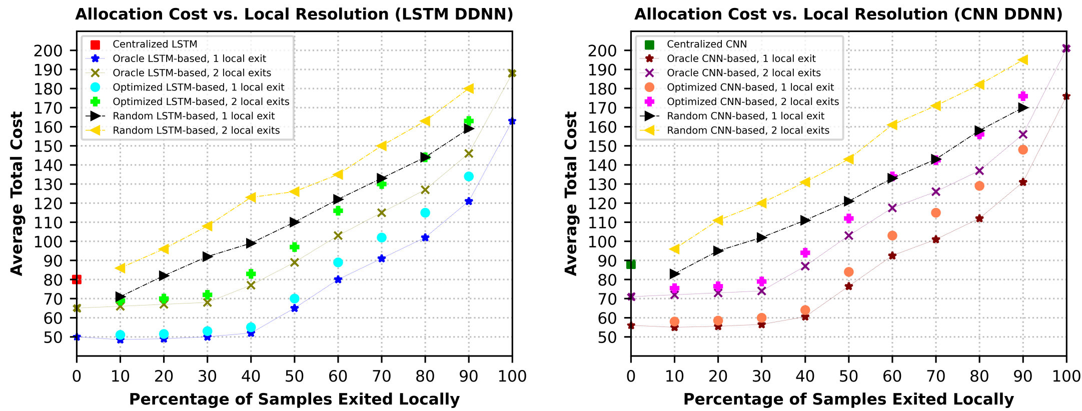
In both Figs. 5.1a and 5.1b, we observe that both offloading mechanisms, confidence-based and Optimized, perform nearly optimally when up to 40% of samples are resolved locally. This demonstrates the effectiveness of both approaches in less demanding scenarios, where a significant portion of samples can be processed at the edge without a substantial increase in total cost. However, as the proportion of locally processed samples increases beyond this point, a noticeable deviation from the optimal Oracle-based bound is observed in both methods.

Notably, the optimized offloading mechanism, which is theoretically derived, consistently outperforms the heuristic confidence-based approach across all scenarios and operating points. This is expected, as the optimized scheme is explicitly designed to minimize costs, leveraging the full power of the optimization problem, whereas the confidence-based method relies on heuristic approximations that are more prone to errors in challenging situations.

Key Observation 3: The ability to make informed online decisions regarding which samples to process locally and how many to handle at the edge introduces significant performance trade-offs. Crucially, these mechanisms allow up to 40% of decisions to be resolved locally "for free", that is, *without increasing the overall provisioning cost when compared to fully centralized models.*

Key Observation 4: The optimized offloading mechanism consistently outperforms heuristics in all scenarios, as expected, since it is derived from a solution to the optimization problem that the oracle already knows.

Figs. 5.2a and 5.2b demonstrate that the LSTM-based DDNN with a single local exit consistently outperforms its CNN-based counterpart. As the proportion of locally exited samples increases, the LSTM-based model with two local exits also tends to maintain lower costs compared to the CNN-based model. This indicates that LSTM-based models may be better suited for scenarios requiring multiple local exits, likely due to their inherent ability to handle sequential data more efficiently. LSTMs are designed to capture



(a) LSTM-based DDNN with 1 and 2 local exits (b) CNN-based DDNN with 1 and 2 local exits

Figure 5.3: Trade-off curves (Total loss vs Percentage of samples predicted locally)

temporal dependencies, making them particularly effective in time-series forecasting tasks, such as resource allocation in dynamic network environments.

Furthermore, the performance gap between Oracle and Optimized offloading mechanisms in CNN-based DDNNs is approximately 5-15% greater than that observed in LSTM-based models. This suggests that LSTM models may inherently manage offloading decisions more effectively, potentially because they leverage the sequential nature of traffic data more efficiently, resulting in more accurate local predictions and minimizing the need for remote resolution.

Figs. 5.3a and 5.3b show that introducing a second local exit generally shifts the trade-off curve upwards, reflecting higher costs across all configurations when compared to models with a single local exit. Additionally, the performance gap between Oracle-based and optimized mechanisms slightly widens with the addition of a second exit. This suggests that while adding more local exits increases the model's capacity to handle samples locally, it also adds complexity to the decision-making process, making it more challenging to balance local and remote processing efficiently.

Key Observation 5: LSTM models are particularly well-suited for applications requiring complex, time-dependent processing. Their ability to capture temporal dependencies

allows them to make more informed decisions when processing sequential data, such as network traffic patterns.

Key Observation 6: Adding a second local exit in parallel increases the complexity of the offloading process, leading to higher overall costs but also providing more opportunities for local processing. While additional local exits offer flexibility in handling samples at the edge, they introduce new challenges in optimizing the balance between local and remote processing, as well as managing the quality of intermediate features. This trade-off must be carefully considered when designing distributed architectures, as increased complexity can diminish the benefits of additional local exits if not properly managed.

In a scenario with two local exits, even if one local exit produces acceptable results and the other does not, both sets of intermediate data from the exits must still be sent to the remote component for final decision-making. This is due to the current design, which treats the local exits collectively rather than evaluating them independently. Consequently, the potential gains from a successful local decision may be diminished when both local outputs are transmitted to the remote layer for further processing.

Although we have not yet implemented an offloading mechanism that independently evaluates the performance of each local exit, this remains a promising area for future exploration. Developing such a mechanism would allow for selective offloading, where samples processed satisfactorily by one local exit are resolved at the edge, while only the unsatisfactory samples from the other exit are forwarded to the remote layers. This approach could significantly reduce overall costs and processing time by eliminating unnecessary transmission of locally resolved samples.

Additionally, introducing a separate cost metric for each local exit in a multi-exit setup could provide more granular insights into the trade-off between local and remote processing. By assigning different weights or costs to the results of each exit, the model could better capture the varying levels of confidence or quality across the exits. This more nuanced approach could influence the trade-off curves, potentially leading to more

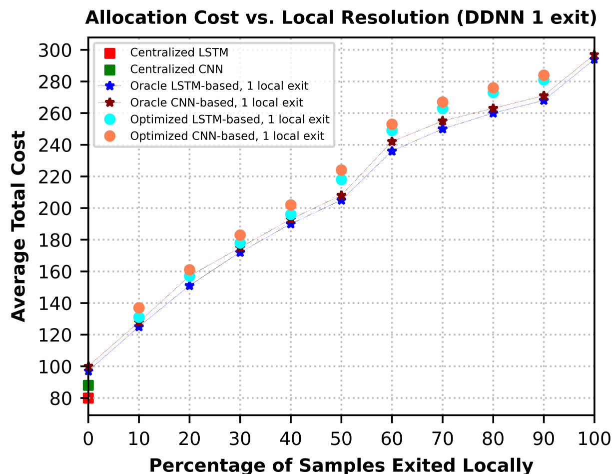


Figure 5.4: Trade-off curve for single local exit DDNN, $(w_L, w_R) = (0.1, 0.9)$

efficient performance by prioritizing exits that yield higher-quality results, while reducing reliance on the remote component.

Exploring independent evaluation and cost metrics for local exits offers a new perspective on optimizing distributed models, and could lead to significant improvements in resource allocation decisions, particularly in scenarios where computational resources are limited or latency is critical.

In Fig. 5.4, where the training weights are set to $(w_L, w_R) = (0.1, 0.9)$, indicating very low strength on local layers (unlike in Figs. 5.1 and 5.2, where higher weights are assigned to local layers), the significant impact of weight selection on overall DDNN performance is evident. While pinpointing the optimal weight pair (in this case the pair $(0.9, 0.1)$) cannot be predetermined, our analysis across various scenarios consistently demonstrates the necessity of assigning higher weights to the local exit for optimal functioning.

Key Observation 7: The selection of training weights (w_L, w_R) plays a pivotal role in the performance of the model. The balance between local and remote processing weights directly influences the DDNN's ability to make quick, effective decisions, showcasing the trade-off between computation speed and accuracy.

Fig.5.5 presents the trade-off curve for the model incorporating four local exits. It

is important to note that the overall size of the model remains constant, with the only change being the addition of more local components of the same size. Despite these changes, the results in Fig.5.5 indicate that our DDNN model with four local exits does not outperform the centralized models at any point along the trade-off curve. This finding suggests that simply adding more local exits does not necessarily lead to improved performance.

We suspect that this lack of improvement may be due to the model size. Specifically, the fixed model size likely limits the effectiveness of each local exit. Since the available resources are divided among more components, each local exit processes a smaller portion of the input data compared to models with fewer local exits. As a result, the quality of the intermediate features generated by each local component diminishes, which negatively impacts the overall performance of the model. Furthermore, distributing data across multiple local exits reduces the amount of information each component can leverage, further degrading the performance compared to centralized models or models with fewer local exits.

Key Observation 8: Adding parallel local components in an attempt to improve performance may not be effective, particularly when the overall model size is fixed. We hypothesize that the model size plays a critical role in determining how many local exits can be effectively incorporated. When local exits are added without increasing the model's overall capacity, each component processes less data, resulting in poorer intermediate features and diminished performance.

5.3.2 SLA Violations Avoidance

Figs. 5.6 and 5.7 illustrate the actual demand ($\mathbf{d}$), local allocations ($\hat{\mathbf{y}}_L$), and remote allocations ($\hat{\mathbf{y}}_R$) for one of the base stations in the dataset under two distinct training weight configurations: $(w_L, w_R) = (0.9, 0.1)$ and $(w_L, w_R) = (0.1, 0.9)$ for the model with one local exit, and $(w_{L1}, w_{L2}, w_R) = (0.9, 0.9, 0.1)$ and $(w_{L1}, w_{L2}, w_R) = (0.1, 0.1, 0.9)$ for

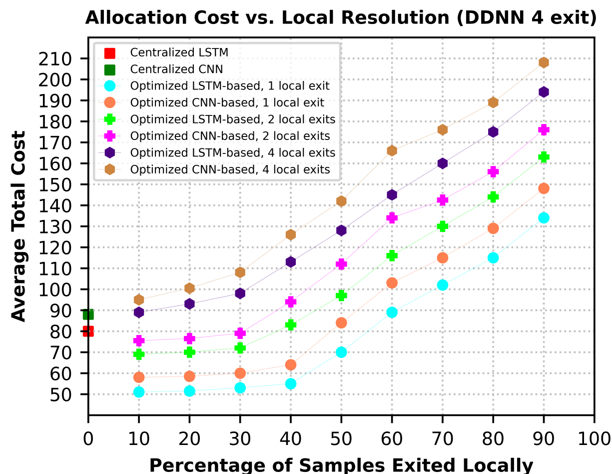
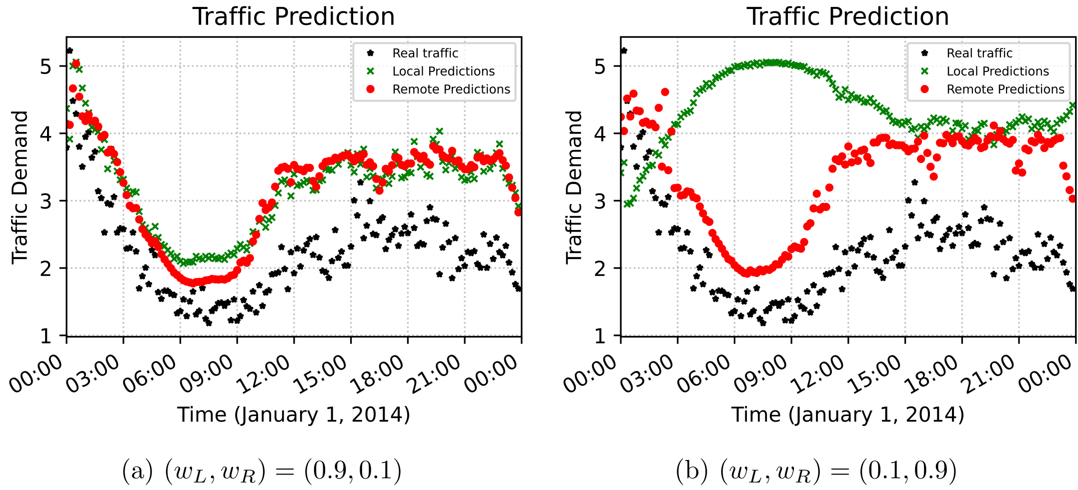


Figure 5.5: Trade-off curve for DDNN with four local exits

the model with two local exits. In order to make a direct comparison of local and remote processing capabilities without interference from the offloading mechanism, the offloading block is deactivated for this analysis.

As the figures demonstrate, configurations with higher local weights, such as $(w_L, w_R) = (0.9, 0.1)$ or $(w_{L1}, w_{L2}, w_R) = (0.9, 0.9, 0.1)$, demonstrates effective local processing performance, allowing a significant portion of the resource allocations to be efficiently handled at the edge. Conversely, the configurations with lower local weights, such as $(w_L, w_R) = (0.1, 0.9)$ or $(w_{L1}, w_{L2}, w_R) = (0.1, 0.1, 0.9)$, result in reduced local processing efficiency and higher costs for local allocations. This trend mirrors the observations from the trade-off curve in Fig. 5.4, where lower local weights increased reliance on remote processing, driving up overall costs. These results highlight the critical impact of weight configurations on the operational efficiency and cost-effectiveness of DDNNs.

Key Observation 9: It is essential to prioritize the local weight over the remote weight (i.e., ensuring $w_L > w_R$ or $w_{L1}, w_{L2} > w_R$) in order to offset the relative simplicity and shallowness of the local module. When local layers are properly weighted, the DDNN can more effectively handle resource allocations at the edge, enabling it to outperform centralized DNN architectures. This highlights the importance of weight selection in



(a) $(w_L, w_R) = (0.9, 0.1)$ (b) $(w_L, w_R) = (0.1, 0.9)$

Figure 5.6: Traffic demand predictions for a base station using two weight pairs, depicted in the scenario without an offloading mechanism

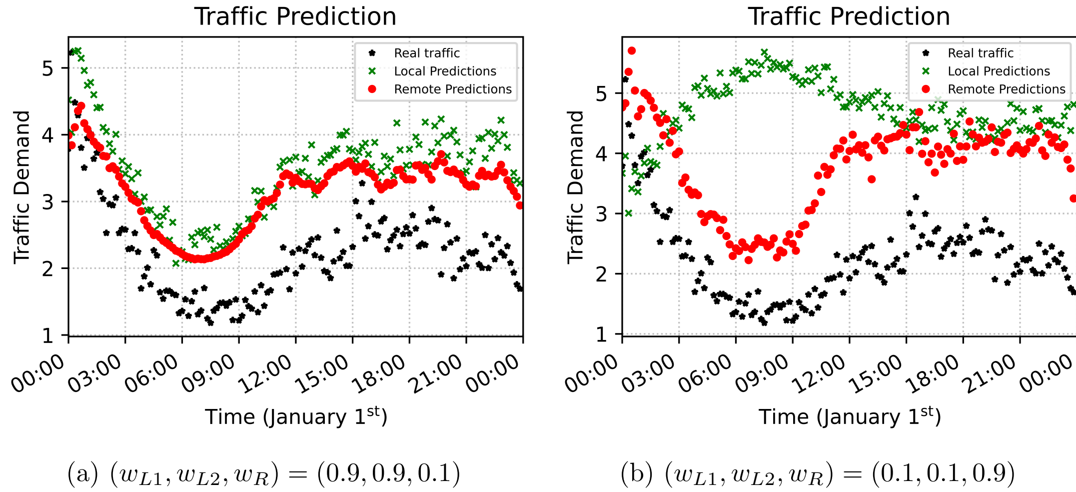
determining the performance of distributed models.

Key Observation 10: The models prioritize preventing SLA violations over precisely matching demand, which aligns with the objective function's emphasis on avoiding under-provisioning and minimizing the cost of over-provisioning rather than merely optimizing for mean squared error (MSE). As seen in the allocation results, the models are biased towards ensuring that there is no under-provisioning, even if it means over-allocating resources.

5.3.3 Latency Reduction

As explained in chapter 4, we evaluate whether our model reduces or increases latency by analyzing both "communication" and "computation" times. For **communication time**, we refer to a recent systems-oriented study [51], which estimates the average round-trip transmission time (RTT) from edge to cloud at 42.46 ms per sample. **Computation time** is assessed by running each model multiple times on the same server and calculating the average processing time per sample across various models[4].

[4]It is important to note that the implementation setup for such a distributed architecture can vary. We designed our experiment using practical values to demonstrate the potential for latency reduction



(a) $(w_{L1}, w_{L2}, w_R) = (0.9, 0.9, 0.1)$ (b) $(w_{L1}, w_{L2}, w_R) = (0.1, 0.1, 0.9)$

Figure 5.7: Traffic demand predictions for a base station using two weight pairs: Data forwarded without offloading mechanism

Table 5.2 presents the results, where "L" represents the percentage of samples processed locally, and "T" is the average time taken to resolve a single sample under each scenario. For example, when 40% of samples are processed locally (i.e., "L" = 40%), the total average resolution time ("T") is calculated by summing:

- The average processing time per sample for the local DNN,
- The average processing time per sample for the offloading block,
- 40% of the average local inference time per sample,
- 60% of the round-trip transmission (RTT) time from edge to cloud and back,
- 60% of the average remote inference time per sample.

The data clearly demonstrate that as the percentage of samples resolved locally increases, the inference latency decreases. Specifically, processing 50% of samples locally with the DDNN not only matches the performance of centralized models in terms of cost efficiency but also achieves a significant 49% reduction in inference latency. This

effectively. The models are executed on a server equipped with an Nvidia V100 GPU, 16 GB HBM2, and 32 GB RAM.

reduction is largely driven by minimizing reliance on cloud-based remote processing, which incurs higher communication costs due to RTT.

Furthermore, the results show that CNN-based DDNN models perform better than LSTM-based DDNN models in terms of latency. CNNs, with their focus on parallelized computations over spatial data, inherently have lower computation times compared to LSTMs, which process data sequentially and require more time to capture temporal dependencies. While both architectures benefit from local exits in reducing latency, CNNs provide a more time-efficient solution.

It is important to note that preprocessing time is not included in this table, as it varies depending on the specific implementation but does not significantly affect the comparison of inference times between the models. The findings emphasize the potential of distributed architectures to not only improve cost efficiency but also significantly reduce latency, especially when a considerable portion of the processing is handled at the edge.

We report the average processing time per sample, and while incorporating parallel local modules does not reduce the latency for individual samples, it can significantly reduce the total processing time for a batch of samples. When each local module processes half of the batch and both produce satisfactory resource allocations, the overall processing time for the batch is effectively halved due to the parallelized operations of the local exits.

When utilizing multiple local exits, the processing time decreases because of the parallel execution of local modules. However, our primary focus is on inference time, which refers to the time required for decision-making. In a model with two local exits, three distinct scenarios can arise:

- **Both local modules produce acceptable allocations:** In this case, the collective inference time for the batch of samples is reduced compared to both the centralized DNN and the DDNN with a single local exit, as decisions are made in parallel by

Table 5.2: Latency Comparison (milliseconds per sample)

L (% of local resolution)	0	5	20	40	50	60	80	95	100
LSTM-based DDNN	42.72	40.40	34.05	25.43	21.09	16.78	8.20	1.75	1.25
CNN-based DDNN	42.70	40.36	34.01	25.37	21.02	16.70	8.10	1.65	1.25
LSTM-based DDNN (Confidence)	42.81	41.01	34.62	26.11	21.86	17.61	9.10	2.72	2.30
Centralized LSTM	42.67	-	-	-	-	-	-	-	-
Centralized CNN	42.63	-	-	-	-	-	-	-	-

the two local modules.

- **Neither local module produces acceptable allocations:** In this scenario, both local modules must send their data to the remote module for further processing, increasing the overall latency as the system waits for remote resolution.

- **One local module produces acceptable allocations while the other does not:** Even if one local module is capable of making a decision, both must transmit their data to the remote module and **wait** for the remote processing to complete. This scenario highlights a limitation of the current offloading mechanism, where the system waits for remote processing even when one local module is already capable of resolving the allocation.

This third scenario, where one local module produces good allocations but must still wait for the remote module, presents an opportunity for improvement. A more sophisticated offloading mechanism could be developed to evaluate each local exit independently, allowing the local module that has generated acceptable results to finalize the decision without waiting for the remote component. Such a mechanism could significantly reduce latency by allowing quicker local decision-making, particularly in cases where only one of the local modules needs to communicate with the remote module. This remains an area for future exploration in optimizing parallel local exits for distributed architectures.

Key Observation 11: The average inference latency diminishes as more samples are processed locally.

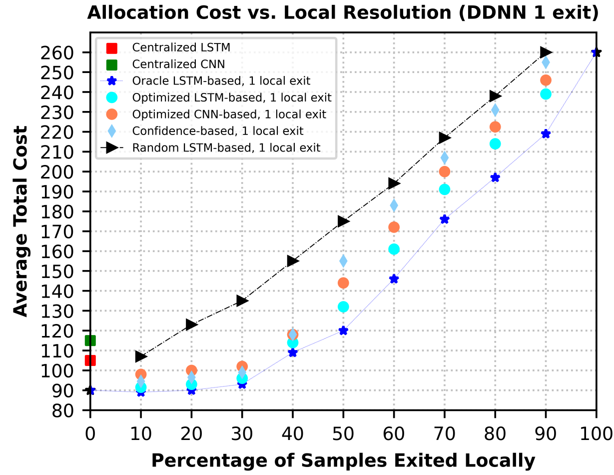


Figure 5.8: Trade-off curve for the model with $(w_L, w_R) = (0.9, 0.1)$ using 25 base stations

5.3.4 Input Size

In our research, we initially employed 16 base stations ($K = 16$) for experimentation. To rigorously test the scalability and responsiveness of our DDNN, we expanded the framework to incorporate 25 base stations ($K = 25$). This expansion aimed to simulate a more demanding network environment, challenging the DDNN with double the initial input size to assess its performance under scaled-up conditions. The results, depicted in Fig. 5.8, demonstrate that the DDNN is able to maintain its robust performance even with the larger number of base stations. Notably, the optimized offloading mechanism employed within our DDNN closely aligns with the Oracle-based optimal policy, underscoring its efficacy and adaptability across a broader input scale.

Key Observation 12: The Distributed DNN architecture exhibits remarkable adaptability, efficiently handling varying input sizes while consistently maintaining targeted performance levels.

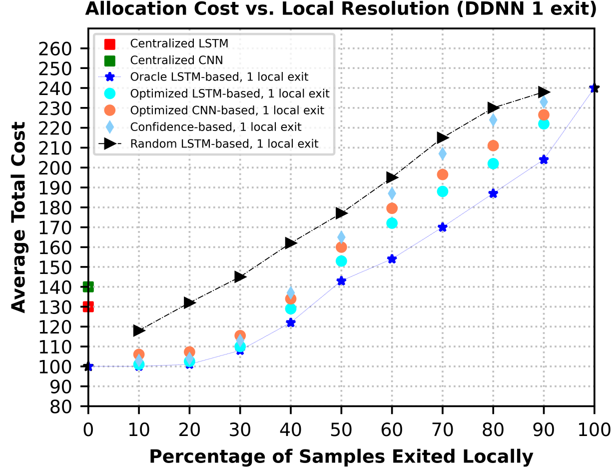


Figure 5.9: Trade-off curve for the model with $(w_L, w_R) = (0.9, 0.1)$ using the objective function in Eq. (2.3) with $c = 10$ and $\epsilon = 0.1$

5.3.5 Objective Function

In our study, we initially applied the objective function as defined in Eq. (2.5) for experimental purposes. To further examine the flexibility of our model, we introduced an additional objective function, as specified in Eq. (2.3). This adjustment was made to evaluate the DDNN's capacity to effectively handle varying objective functions and assess its adaptability to different cost models and constraints. The outcomes, illustrated in Fig. 5.9, confirm that the DDNN maintains stable and robust performance under these adjusted conditions. Crucially, the optimized offloading mechanism within the DDNN framework demonstrated its versatility by continuing to closely align with the optimal policy, even after modifying the objective function.

Key Observation 13: The DDNN exhibits a high degree of adaptability to different objective functions while maintaining near-optimal performance. This flexibility ensures that the DDNN can be deployed across various application scenarios with distinct cost models or performance criteria, without compromising its efficiency or decision-making accuracy.

Chapter 6

Future Work and Conclusions

6.1 Future Work

Advanced Offloading Methods: Future work will explore more advanced offloading strategies, ranging from simple estimations to sophisticated reinforcement learning approaches. These methods would be designed to handle scenarios where the cost C_T (such as transmission cost or latency) is dynamic and unknown, allowing the system to adapt in real time and optimize performance in fluctuating network conditions.

Joint Training of Offloading: Another promising area for future research involves the simultaneous training of the offloading decision mechanism alongside the DDNN for cost minimization. By integrating offloading decisions with the training process, the model can leverage shared features across specific local layers to achieve more efficient and accurate offloading while minimizing overall system costs.

Weight Scheduling: Investigating adaptive strategies for adjusting the local and remote weights (w_L, w_R) during the DDNN training process is an area for future exploration. Dynamic weight scheduling could optimize model performance by adapting to varying network demands and system constraints over time, further improving the balance between local and remote processing.

Scaling and Performance in Larger Networks: In future work, we plan to scale up our experimental models to assess the architecture's performance in larger and more complex scenarios. Evaluating the DDNN's efficacy in handling larger-scale networks with significantly more base stations and traffic variability will provide valuable insights into the architecture's practical applicability in real-world environments.

Independent Offloading Decisions for Local Exits: Another intriguing direction for future research is the exploration of independent offloading decisions for each local exit. Currently, offloading decisions are made collectively, which may result in unnecessary data transmission to the remote component. Allowing local exits to make independent decisions, based on their individual performance, could reduce latency and improve overall resource efficiency, especially in scenarios where only one of the local exits provides an acceptable allocation.

6.2 Conclusions

In this study, we developed and implemented a Distributed Deep Neural Network (DDNN) tailored to forecast future traffic demand and optimize resource allocation in 5G networks. The proposed DDNN features a multi-exit architecture, incorporating both local exits (e.g., at the network edge) and a remote exit (e.g., in the cloud). This design allows for efficient decision-making at various levels of the network, balancing the trade-off between latency and processing power. To ensure optimal performance, the DDNN undergoes joint training, where specific weights are assigned to the local and remote exits, enabling the model to distribute the decision-making load effectively.

Our results indicate that the DDNN architecture maintains performance at lower or comparable costs to centralized models, while resolving nearly 50% of resource allocation decisions locally. This substantial local processing capability reduces overall system latency and communication overhead, making the model highly efficient for real-time applications. In particular, LSTM-based models demonstrated superior performance

over CNN-based models for handling time-series data, such as traffic demand forecasting, making them particularly well-suited for this task.

Furthermore, we showed that a well-designed offloading mechanism, which decides which samples should be processed locally and which should be sent to the cloud, can significantly enhance performance. This mechanism consistently outperformed existing methods, bringing the model's performance closer to that of the oracle-based optimal policy. The DDNN's ability to make near-optimal offloading decisions highlights its robustness and potential for deployment in large-scale, dynamic 5G network environments.

In conclusion, the DDNN framework provides an efficient and scalable solution for traffic forecasting and resource allocation in 5G networks. By leveraging its multi-exit architecture and the offloading mechanism, it achieves substantial reductions in latency and cost, while maintaining high performance.

Appendices

Appendix A

Chapter 3 Appendices

A.1 Prediction Uncertainty

In our framework, we represent a neural network as a function denoted as $\mathcal{F}(\cdot;\boldsymbol{\theta})$, where $\mathcal{F}$ encapsulates the network architecture, and $\boldsymbol{\theta}$ represents the set of model parameters. Within the context of a Bayesian neural network, we introduce a prior distribution for the weight parameters, and the objective is to effectively model the posterior distribution. Commonly, a Gaussian prior, $\boldsymbol{\theta} \sim N(0, I)$, is employed. Additionally, we specify the data generating distribution $p(y|\mathcal{F}(d;\boldsymbol{\theta}))$. In regression tasks, it is often assumed that $y|\boldsymbol{\theta}$ follows a normal distribution, i.e., $y|\boldsymbol{\theta} \sim N(\mathcal{F}(d;\boldsymbol{\theta}), \sigma^2)$, with σ representing the noise level.

In Bayesian inference, we assume a dataset comprising N observations, denoted as $\mathbf{d} = \{d_1, ..., d_N\}$, and their corresponding outcomes, represented as $\mathbf{y} = \{y_1, ..., y_N\}$. The primary objective is to derive the posterior distribution over the model parameters, which is expressed as $p(\boldsymbol{\theta}|\mathbf{d}, \mathbf{y})$. When we introduce a new data point, d^*, the prediction distribution is acquired through the process of marginalization over the posterior distribution:

$$p(y^*|d^*) = \int_{\boldsymbol{\theta}} p(y^*|\mathcal{F}(d^*;\boldsymbol{\theta}))p(\boldsymbol{\theta}|\mathbf{d},\mathbf{y})d\boldsymbol{\theta}, \tag{A.1}$$

Specifically, the prediction distribution's variance serves as a measure of prediction uncertainty, and this uncertainty can be subjected to further decomposition employing the law of total variance:

$$\begin{aligned} \text{Var}(y^*, d^*) &= \text{Var}[E(y^*|\boldsymbol{\theta}, d^*)] + E[\text{Var}(y^*|\boldsymbol{\theta}, d^*)] \\ &= \text{Var}(\mathcal{F}(d^*;\boldsymbol{\theta})) + \sigma^2, \end{aligned} \tag{A.2}$$

We break down the variance into two key parts: (i) $\text{Var}(\mathcal{F}(d^*;\boldsymbol{\theta}))$, which shows how unsure we are about the model's parameters, known as *model uncertainty*; and (ii) σ^2, the noise level that naturally occurs in the process of creating our data, known as *inherent noise*.

Model uncertainty plays a crucial role in understanding how confident we are in our neural network's predictions. Estimating this uncertainty involves computing the posterior distribution $p(\boldsymbol{\theta}|\mathbf{d},\mathbf{y})$, utilizing a process called Bayesian inference. However, this task becomes complex in neural networks because of the non-linear nature of the models, leading to what's known as non-conjugacy. Several studies have explored methods for approximating this inference process in the context of deep learning. Our approach, inspired by research in [42], [43], and [52], employs Monte Carlo dropout (MCdropout). MCdropout is a practical method for approximating model uncertainty by randomly deactivating some neurons, which helps simulate the effects of sampling from the posterior distribution.

The algorithm leverages stochastic dropout to estimate model uncertainty as follows: Given a new input x^*, the neural network's output is calculated multiple times, each with stochastic dropouts applied at every layer. For this purpose, random dropout is applied to each hidden unit with a predefined probability p, creating variations in the network's

architecture during each forward pass. This stochastic feed-forward process is repeated J times, generating a diverse set of outputs $\{\hat{y}_1^*, ..., \hat{y}_J^*\}$, which reflect the variability introduced by dropout. The sample variance of these outputs is then calculated, serving as an approximation of the model's uncertainty.

$$\widehat{\mathrm{Var}}(\mathcal{F}(d^*; \boldsymbol{\theta})) = \frac{1}{J} \sum_{j=1}^{J} (\hat{y}_j^* - \overline{\hat{y}^*})^2, \tag{A.3}$$

where $\overline{\hat{y}^*} = \frac{1}{J} \sum_{j=1}^{J} \hat{y}_j^*$.

Bibliography

[1] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2224–2287, 2019.

[2] C.-X. Wang, M. D. Renzo, S. Stanczak, S. Wang, and E. G. Larsson, "Artificial intelligence enabled wireless networking for 5g and beyond: Recent advances and future challenges," *IEEE Wireless Communications*, vol. 27, no. 1, pp. 16–23, 2020.

[3] D. Praveen Kumar, T. Amgoth, and C. S. R. Annavarapu, "Machine learning algorithms for wireless sensor networks: A survey," *Information Fusion*, vol. 49, pp. 1–25, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156625351830277X

[4] M. Chen, U. Challita, W. Saad, C. Yin, and M. Debbah, "Artificial neural networks-based machine learning for wireless networks: A tutorial," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3039–3071, 2019.

[5] Q. Liu, T. Han, and E. Moges, "Edgeslice: Slicing wireless edge computing network with decentralized deep reinforcement learning," 2020. [Online]. Available: https://arxiv.org/abs/2003.12911

[6] V. Sciancalepore, X. Costa-Perez, and A. Banchs, "Rl-nsb: Reinforcement learning-

based 5g network slice broker," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, p. 1543–1557, aug 2019. [Online]. Available: https://doi.org/10.1109/TNET.2019.2924471

[7] Y. Liu, J. Ding, and X. Liu, "A constrained reinforcement learning based approach for network slicing," in *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, 2020, pp. 1–6.

[8] H. Halabian, "Distributed resource allocation optimization in 5g virtualized networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 627–642, 2019.

[9] C. Zhang, M. Fiore, C. Ziemlicki, and P. Patras, "Microscope: mobile service traffic decomposition for network slicing as a service," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372224.3419195

[10] J. X. Salvat, L. Zanzi, A. Garcia-Saavedra, V. Sciancalepore, and X. Costa-Perez, "Overbooking network slices through yield-driven end-to-end orchestration," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 353–365. [Online]. Available: https://doi.org/10.1145/3281411.3281435

[11] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Deepcog: Cognitive network management in sliced 5g networks with deep learning," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 280–288.

[12] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks

over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.

[13] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Aztec: Anticipatory capacity allocation for zero-touch network slicing," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 794–803.

[14] A. T. Z. Kasgari and W. Saad, "Stochastic optimization and control framework for 5g network slicing with effective isolation," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, 2018, pp. 1–6.

[15] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "Robust user association for ultra dense networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2690–2698.

[16] N. Liakopoulos, A. Destounis, G. Paschos, T. Spyropoulos, and P. Mertikopoulos, "Cautious regret minimization: Online optimization with long-term budget constraints," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 3944–3952. [Online]. Available: https://proceedings.mlr.press/v97/liakopoulos19a.html

[17] N. Liakopoulos, G. Paschos, and T. Spyropoulos, "No regret in cloud resources reservation with violation guarantees," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1747–1755.

[18] A. Okic, L. Zanzi, V. Sciancalepore, A. Redondi, and X. Costa-Pérez, "π-road: a learn-as-you-go framework for on-demand emergency slices in v2x scenarios," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[19] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1709.01686

[20] Y. Kaya, S. Hong, and T. Dumitras, "Shallow-deep networks: Understanding and mitigating network overthinking," 2019. [Online]. Available: https://arxiv.org/abs/1810.07052

[21] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, "Why should we add early exits to neural networks?" *Cognitive Computation*, vol. 12, no. 5, p. 954–966, Jun. 2020. [Online]. Available: http://dx.doi.org/10.1007/s12559-020-09734-4

[22] S. P. Chinchali, E. Cidon, E. Pergament, T. Chu, and S. Katti, "Neural networks meet physical networks: Distributed inference between edge devices and the cloud," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 50–56. [Online]. Available: https://doi.org/10.1145/3286062.3286070

[23] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372224.3419194

[24] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1423–1431.

[25] H. N. N. U, M. K. Hanawal, and A. Bhardwaj, "Unsupervised early exit in dnns with multiple exits," 2022. [Online]. Available: https://arxiv.org/abs/2209.09480

[26] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 615–629, apr 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037698

[27] M. Sbai, N. Trigoni, and A. Markham, "Multiple early-exits strategy for distributed deep neural network inference," in *2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2023, pp. 34–38. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DCOSS-IoT58021.2023.00014

[28] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense networks for resource efficient image classification," 2018. [Online]. Available: https://arxiv.org/abs/1703.09844

[29] O. Nassef, W. Sun, H. Purmehdi, M. Tatipamula, and T. Mahmoodi, "A survey: Distributed machine learning for 5g and beyond," *Computer Networks*, vol. 207, p. 108820, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622000421

[30] X. Chen, C. Wu, Z. Liu, N. Zhang, and Y. Ji, "Computation offloading in beyond 5g networks: A distributed learning framework and applications," *IEEE Wireless Communications*, vol. 28, no. 2, pp. 56–62, 2021.

[31] T. Giannakas, T. Spyropoulos, and O. Smid, "Fast and accurate edge resource scaling for 5g/6g networks with distributed deep neural networks," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022, pp. 100–109.

[32] J. Wang, J. Tang, Z. Xu, Y. Wang, G. Xue, X. Zhang, and D. Yang, "Spatiotemporal modeling and prediction in cellular networks: A big data enabled deep learning ap-

proach," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[33] C. Zhang and P. Patras, "Long-term mobile traffic forecasting using deep spatio-temporal neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1712.08083

[34] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Comput. Surv.*, vol. 55, no. 5, dec 2022. [Online]. Available: https://doi.org/10.1145/3527155

[35] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," 2014. [Online]. Available: https://arxiv.org/abs/1409.5185

[36] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014. [Online]. Available: https://arxiv.org/abs/1409.4842

[37] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," 2018. [Online]. Available: https://arxiv.org/abs/1810.06060

[38] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2017. [Online]. Available: https://arxiv.org/abs/1610.05492

[39] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," 2014. [Online]. Available: https://arxiv.org/abs/1409.5185

[40] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, "Deebert: Dynamic early exiting for accelerating bert inference," 2020. [Online]. Available: https://arxiv.org/abs/2004.12993

[41] P. M. Grulich and F. Nawab, "Collaborative edge and cloud neural networks for real-time video processing," *Proc. VLDB Endow.*, vol. 11, no. 12, p. 2046–2049, aug 2018. [Online]. Available: https://doi.org/10.14778/3229863.3236256

[42] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," 2016. [Online]. Available: https://arxiv.org/abs/1506.02142

[43] L. Zhu and N. Laptev, "Deep and confident prediction for time series at uber," in *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, Nov. 2017. [Online]. Available: http://dx.doi.org/10.1109/ICDMW.2017.19

[44] Telecom Italia, "Milano grid," 2015. [Online]. Available: https://doi.org/10.7910/DVN/QJWLFU

[45] J. Paparrizos and L. Gravano, "k-shape: Efficient and accurate clustering of time series," *SIGMOD Rec.*, vol. 45, no. 1, p. 69–76, jun 2016. [Online]. Available: https://doi.org/10.1145/2949741.2949758

[46] I. Borg and P. Groenen, "Modern multidimensional scaling: Theory and applications," *Journal of Educational Measurement*, vol. 40, pp. 277 – 280, 06 2006.

[47] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, Mar. 1955.

[48] A. Furno, M. Fiore, R. Stanica, C. Ziemlicki, and Z. Smoreda, "A tale of ten cities: Characterizing signatures of mobile traffic in urban areas," *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2682–2696, 2017.

[49] R. G. Pacheco, R. S. Couto, and O. Simeone, "Calibration-aided edge inference offloading via adaptive model partitioning of deep neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2010.16335

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[51] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, "Couper: Dnn model slicing for visual analytics containers at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 179–194. [Online]. Available: https://doi.org/10.1145/3318216.3363309

[52] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1512.05287