

Imperial College of Science, Technology and Medicine
Department of Computing

Combining Queueing Analysis and Machine Learning for Performance Modeling of Distributed Systems

Zifeng Niu

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy of Imperial College London
September 2024

Statement of Originality

I declare that this thesis was written by myself, and the presented work is my own, except stated otherwise.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

Performance modeling is commonly used during the early design phase of a system to ensure that performance meets quality of service (QoS) requirements. Queueing analysis is a classic analytical approach that abstracts queueing networks (QNs) from systems and quantitatively characterizes system behavior to support decision-making. However, most realistic QNs are mathematically intractable, leading to two key challenges. The first challenge is characterizing complex response time distributions, a task beyond the capabilities of conventional approaches. The second challenge is accurately evaluating the performance of non-product-form QNs, for which queueing analysis lacks a closed-form solution. Machine learning (ML) advancement presents new opportunities for performance modeling. Nevertheless, the use of ML faces the challenge of lacking flexibility, making it difficult to apply to QNs with varying topologies and numbers of job classes, which limits its potential as a general-purpose solution.

We address these challenges by combining queueing analysis and ML approaches. To estimate response time distributions, we propose a phase-type density-passing framework which recursively propagates response time distributions predicted by mixture density networks, enabling accurate approximations. To address the challenges of evaluating non-product-form QNs, we leverage insights from queueing analysis to design graph neural network (GNN) models. First, we develop a model to solve closed multiclass QNs. Similar to analytical methods, our GNN predicts performance measures per job class at each service center. This exploration reveals the potential of a learning-based model as a general-purpose solver for various QNs. Next, we define a more fine-grained graph for open multiclass QNs abstracted from distributed edge AI systems and customize message passing based on queueing analysis of chain-based services. To broaden its applicability, we propose a solution for achieving out-of-distribution generalization ability. We demonstrate that the proposed model is an accurate yet efficient performance modeling tool for determining the optimal plan during the system design.

Acknowledgements

Time flies when you are enjoying the journey. I am near the end of my PhD and I would like to express my heartfelt gratitude to those who make my memory so valuable and special.

From the bottom of my heart, I want to say a big thank you to my supervisor, Giuliano Casale. He is the ideal supervisor for me, and it has been an honor to have the chance to work with him. Giuliano is completely dedicated, enthusiastic, patient, and supportive. I am profoundly grateful to him for guiding me in the subject of performance modeling. Over the past few years, I have learned more than I could have imagined, and I am thankful for all the knowledge I have gained from Giuliano. His wealth of expertise and experience is like a vast ocean, while I am merely a small pond in comparison. Under his guidance, I gained many precious first-time experiences, such as writing my first paper, publishing my first journal article, and presenting at my first conference. Giuliano has a deep understanding of queueing theory as well as modern technological trends, and the directions he suggests often lead to insightful research outcomes. This is because he places great importance on learning, his enthusiasm and persistence in learning new topics will inspire me for a long time. Not only is Giuliano a great supervisor, but he is also a warm and trustworthy friend. He cares about his team members not only in terms of their academic progress but also their personal well-being. He always does his best to provide us with help and support. I would again like to convey my deepest gratitude and thanks to my supervisor and friend, Giuliano Casale.

I would also like to show my deepest appreciation to Manuel Roveri. He is a professor at Politecnico di Milano, leading a motivated and energetic team which I am lucky to visit for two months. They warmly hosted me, making me feel like part of their family. Manuel has conducted an outstanding research on edge intelligence, it has been a fantastic experience collaborating with him. Besides, his qualities of passion and optimism have greatly influenced me. I extend my best wishes to Manuel and all his team members for their continued success.

I would like to appreciate all my colleagues in QORE lab. I have vivid memories of the times we worked and played together. I would like to express my deepest thanks to my parents for their love and steadfast support. I would also like to thank my office, room 420, a cosy study space accompanying me throughout my PhD journey. Finally, I am grateful for this era where we are experiencing unprecedented rapid technology developments and seeing infinite possibilities.

‘System analysis is an art, not a science.’

KM Chandy, CH Sauer (1978)

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Motivation and Research Objectives	1
1.2 Thesis Organization and Contributions	5
1.3 Publications	7
2 Background	10
2.1 Performance Modeling	10
2.2 Basics of Queueing Theory	11
2.2.1 Exponential Distribution and the Poisson Process	11
2.2.2 Markov Chains and Phase-Type Distributions	13
2.2.3 Queueing Networks	17
2.2.4 Layered Queueing Networks	24
2.3 Machine Learning Approaches	26
2.3.1 A General Workflow	26

2.3.2	Mixture Density Networks	27
2.3.3	Graph Neural Networks	29
2.4	Related Work	34
3	Neural Density Estimation of Response Times in Layered Software Systems	38
3.1	Introduction	38
3.2	Prerequisites	39
3.2.1	LQN Layering Strategy	39
3.2.2	Queueing Network Submodels	40
3.2.3	Routing Paths	42
3.2.4	Modeling Requests to Entries	43
3.2.5	Phase-Type Distributions	43
3.2.6	Client Task and Entry Chains	44
3.3	Methodology	46
3.3.1	Overview	46
3.3.2	Class Aggregation	49
3.3.3	Submodel Response Time Predictions	50
3.3.4	PH Models of Activity Graphs	51
3.3.5	End-to-End Response Time PH Computation	55
3.4	Evaluation	56
3.4.1	MDN Submodel Analyzers	56
3.4.2	LQN Response Time Prediction	60
3.4.3	Layering Depth Experiment	64

3.5	Summary	67
4	Graph Representation Learning for Evaluating Closed Queueing Networks	69
4.1	Introduction	69
4.2	Problem Formulation	70
4.3	Methodology	71
4.3.1	Performance Evaluation using MLPs	71
4.3.2	Underlying Motivations for the GNN Design	73
4.3.3	Model Input: Graph Representation and Features	75
4.3.4	The Flow of Updating Node States	76
4.3.5	Message Passing on the Global Graph	76
4.3.6	Message Passing on the Local Graph	77
4.3.7	The GNN Algorithm for Solving CQNs	78
4.4	Evaluation	80
4.4.1	Two-class Machine Repairman Networks	80
4.4.2	General Machine Repairman Networks	81
4.4.3	Closed Multiclass Queueing Networks	84
4.4.4	Case Studies on LQNs and Routing Probabilities	87
4.4.5	Networks with Blocking-After-Service Mechanism	89
4.4.6	The configuration of model parameters	92
4.5	Summary	96

5	Customized Graph Model for Loss-aware Distributed Service Placement	98
5.1	Introduction	98
5.2	Problem Formulation	100
5.3	A Motivating Example	102
5.4	Methodology	103
5.4.1	Workflow	103
5.4.2	Model input: graph representation and features	104
5.4.3	Design rationale based on queueing analysis	105
5.4.4	Model design for a single service chain	107
5.4.5	Overall Algorithm	109
5.5	Generalized GNN for Larger Placements	112
5.5.1	Design for a shared device by multiple execution steps	112
5.5.2	Design for generalization in out-of-distribution scenarios	113
5.6	Surrogate optimization method	115
5.7	Evaluation	116
5.7.1	Experiment setup	117
5.7.2	Evaluating the proposed GNN model	118
5.7.3	Evaluating the surrogate optimization program	123
5.7.4	A case study with real-world parameters	128
5.8	Summary	129
6	Conclusion	130
6.1	Summary of Thesis Achievements	130

6.2 Future Work 133

Bibliography **134**

List of Tables

3.1	Example of routing path for the activity graph associated to <i>Entry2</i> in Figure 2.6. The <i>App Proc</i> is a PS station.	44
3.2	Submodel parameters.	57
3.3	Neural network architecture and optimizer.	58
3.4	Minimum validation loss of MDNs.	58
3.5	The MAPE and MKS results of MDNs.	59
3.6	Case study parameters.	63
3.7	Percentiles of APE distributions and MAPE of standard deviation across 500 instances with topology as in Figure 3.10.	65
3.8	Model size parameters.	66
4.1	Performance measures of class-1 jobs in motivating examples shown in Figure 4.1.	74
4.2	The MAPE and percentiles of the APE distribution of queue length, utilization, response time, and throughput predictions made by AMVA, MLP, and Gsolver.	80
4.3	The percentiles of the APE distribution of queue length, utilization, response time, and throughput predictions made by Gsolver- β and its two variants. . . .	86
4.4	MAPEs, MdAPEs for system performance predictions made by AMVA and Gsolver and their computation times across three LQN groups shown in Fig- ure 4.7.	87

4.5	The MAPEs and percentiles of the APE distribution of system response time and throughput predictions made by AMVA, Gsolver and baseline GNN.	91
4.6	Gsolver parameters.	92
4.7	Absolute percentage errors in predicting two-class machine repairman networks using Gsolver, trained with different numbers of message-passing iterations. . . .	93
4.8	Absolute percentage errors in predicting general machine repairman networks using Gsolver, trained with different numbers of message-passing iterations. . . .	93
4.9	Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different numbers of message-passing iterations and attention heads.	94
4.10	Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different numbers of message-passing iterations.	94
4.11	Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different sizes of hidden layers.	94
4.12	Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different batch sizes.	95
5.1	Main notation used by ChainNet.	110
5.2	Feature modifications for generalization.	115
5.3	Parameters used for network generation.	117
5.4	GNN hyperparameters.	118
5.5	Throughput APE results by ChainNet and baseline models.	118
5.6	MAPE results by ChainNet and its ablated variants.	122
5.7	Parameters used for placement problem generation.	123

List of Figures

1.1	Main developments and corresponding thesis chapters.	6
2.1	The use of performance modeling in the system design process.	11
2.2	An example of a S -phase PH distribution.	16
2.3	The series canonical form of acyclic PH distributions.	17
2.4	Examples of a closed and an open queueing networks.	18
2.5	Little's law applied at three levels.	20
2.6	An example of an LQN model.	26
2.7	Machine learning workflow.	27
2.8	The structure of a Gamma-MDN.	30
2.9	One iteration of message passing for a node.	31
2.10	Unfolding computation graphs to visualize how a node interacts with the surrounding context through the iterations of message passing.	32
2.11	GNN training pipeline.	34
3.1	Task-submodel and Processor-submodel in an SRVN style decomposition. Submodels I and II are single-class closed queueing networks with 10 jobs, representing workers. The queueing network for submodel III is not shown as it is similar to the one of submodel II.	41

- 3.2 A submodel with multiple entries. Each entry is assumed to require a single visit to the processor sharing station modeling host processor P1. The submodel has $N = 10$ jobs, equivalent to the multiplicity of the client task $T1$. The total throughput X_{total} at the lower layer model is equivalent at steady state to the sum of the throughputs X_k^{upper} of each entry $k = 1, \dots, K$ 45
- 3.3 Schematic of applying the proposed method on an LQN model. Circles 1, 2, and 3 represent computing host demands of entries using proposed PH operations, and $\oplus$ represents a convolution operation. $W_{A \rightarrow B}$ denotes the random variable for the response time of entry A at B . $PH_{A \rightarrow B}$ denotes the PH representation of $W_{A \rightarrow B}$ 47
- 3.4 A comparison between simulation and estimated conditional density functions of the response time given an input vector $\mathbf{x} = (\{90, 3, 0.6, 1.25\}, 8)$ with (a) FCFS and (b) PS, respectively. 47
- 3.5 Underlying Markov chain of the loop structure. 51
- 3.6 Loop transformation. 54
- 3.7 (a): The CDFs of KS distances between simulations and predictions for all FCFS and PS test pairs. (b): Summary statistics (the orange line and green triangle inside box indicate the median and mean value) on the response time given $\mathbf{x} = (\{85, 8, 0.4, 1.5\}, 2)$. The left and right y-axis represent the response time under FCFS and PS, respectively. 59
- 3.8 (a)-(b): Mean $\pm$ one standard deviation (shaded area) of the response time. Starting with $\mathbf{x} = (\{90, 8, 1, 0.5\}, 4)$, varying population and number of servers. (c)-(d): Percentiles of the response time. Starting with $\mathbf{x} = (\{35, 8, 1, 0.5\}, 4)$, varying think time and host demand mean. Sim and Pre represent simulation and prediction. 60
- 3.9 (a)-(b): Mean $\pm$ one standard deviation (shaded area) of the response time. Starting with $\mathbf{x} = (\{25, 3, 0.1, 2\}, \{30, 7, 0.2, 0.8\}, 4)$, varying host demand mean of class 2. (c)-(d): Analyzing a submodel $\mathbf{x} = (\{15, 3, 0.7, 1\}, \{25, 10, 0.8, 0.6\}, \{30, 9, 0.1, 1\}, \{20, 5, 0.3, 1.4\}, 2)$ under FCFS. 61

3.10	LQN model of an E-commerce system.	63
3.11	Three activity graphs used in the case study of Figure 3.10.	64
3.12	LQN model of an interconnected system.	64
3.13	MAPE and APE distribution across 500 instances with topology as in Figure 3.12. Here, error bars represent the 95% confidence interval. It can be observed that the confidence intervals are relatively narrow.	65
3.14	(a): Example of a randomly generated LQN model, for ease of presentation, entry and activity details are omitted. (b)-(c): MAPE and MAE over different number of layers. The bar diagram shows 95% confidence intervals. (d): AE of the exceedance probability categorized into three groups based on number of layers.	66
4.1	Motivating examples underlying the rationale for our GNN design.	73
4.2	The graph representation for the motivating two-class queueing network in Figure 4.1. There are four types of elements in a graph representation: (i) global nodes: filled nodes with solid borders; (ii) local nodes: filled nodes with dashed borders (iii) class edges: solid line arrows; (iv) center edges: dashed line arrows.	75
4.3	Schematic view of the state update flow in our GNN approach.	76
4.4	(a): MAPEs of four performance measure predictions on the second group of two-class models. (b): MAPEs of response time predictions in distinct groups classified by the difference between service times of two job classes.	80
4.5	(a): CDFs of the absolute percentage errors in GNN predictions for four performance measures. (b): MAPEs of throughput and response time predictions made by the proposed Gsolver and its two variants. (c): MAPEs of response time predictions from Gsolvers across three groups, each classified by a distinct number of job classes. (d): The distributions of response time predictions and ground truth under different numbers of job classes.	83

4.6	(a): MAPEs for predicting the performance measures of closed multiclass queueing networks using Gsolver- β and its two variants. (b): Box plots showing APEs for throughput predictions by Gsolver- β and Gsolver- γ - Q across different route lengths. (c): The bottleneck is the most utilized resource in the network and serves as the limiting factor for system performance. (d): MAPEs for predicting performance measures of two groups of machine repairman queueing networks using Gsolver- γ - Q	85
4.7	Three groups of LQNs for evaluating the prediction performance of LQN solution algorithms powered by AMVA and Gsolver, respectively.	87
4.8	Two groups of closed queueing networks with routing probabilities for evaluating the prediction performance of AMVA and Gsolver.	88
4.9	MAPEs for performance measure predictions made by AMVA and Gsolver on closed queueing networks (a) and (b) shown in Figure 4.8.	89
4.10	Blocking-after-service mechanism. This mechanism blocks the server when the destination reaches maximum capacity.	90
4.11	Solving closed single-class tandem QNs using off-the-shelf GNN blocks.	91
4.12	(a): CDFs of the absolute percentage errors in system throughput and response time predictions made by Gsolver and baseline GNN. (b): The distributions of response time predictions and ground truth across three groups classified by the lengths of the networks.	92
4.13	The impacts of the number of message-passing iterations, hidden layer size, and batch size on the prediction performance of Gsolver.	95
5.1	An example of deploying three edge AI services on five devices. Both chain-1 and chain-2 services comprise four fragments, while chain-3 service comprises three fragments. Fragments are represented by boxes. The number within a box represents the position of this fragment in its service chain.	103

5.2	The queueing network model of the placement in Figure 5.1. A station with a finite-capacity buffer is represented by a circle followed by a series of rectangles. The arrival rates of requests for three services are λ_1 , λ_2 , and λ_3	104
5.3	Main steps of the proposed approach.	104
5.4	Creating input graph corresponding to the placement decision. There are five types of elements in a graph representation: (i) service nodes: hollow circles with solid red borders; (ii) fragment nodes: filled nodes with solid blue borders; (iii) device nodes: filled nodes with dashed green borders (iv) workflow edges: solid arrows; (v) placement edges: dashed arrows.	105
5.5	The execution sequence of the chain- i service comprising T_i fragments. It is partitioned into execution steps $E_1, \dots, E_{T_i}$. The color scheme follows the one used in Figure 5.4.	106
5.6	The n -th iteration of the proposed message passing within the j -th execution step of the chain- i service. The color scheme follows the one used in Figure 5.4. The colored dash-dotted lines copy node embeddings. The black arrows indicate input/output of the three update functions ϕ_C , ϕ_F , and ϕ_D	107
5.7	Predicting system throughput and end-to-end latency of the chain- i service concurrently at the last (N -th) iteration. The $\oplus$ represents element-wise addition of vectors. The color scheme follows the one used in Figure 5.4.	110
5.8	Message passing for a device node shared by multiple execution steps. The color scheme follows the one used in Figure 5.4.	112
5.9	Two out-of-distribution scenarios considered by our surrogate model.	113
5.10	(a) Suppose the only differences between these two cases are the arrival rate and processing time of the chain-1 service. The modified features of the processing time in the two cases are the same: $t_{p_{1,1}}\lambda_1 = 0.875$ and $t_{p_{2,1}}\lambda_2 = 0.25$. (b) Although all modified features are the same between the two cases, they have different data loss rates of the chain-2 service.	114

5.11	(a)-(b): MAPE results on Type I and II test datasets, respectively. (c)-(d): APE distributions on Type I and II test datasets, respectively.	119
5.12	Box plot of APE distributions under different number of nodes and service chains. The horizontal line that splits the box is the median. We only present the throughput prediction error boxes of ChainNet and GAT in Figure 5.12b and Figure 5.12d. This is because the medians of GIN boxes are higher than the third quartiles of other boxes, making it inconvenient to display them alongside ChainNet and GAT.	121
5.13	Ablation study. Variants α , β , and δ do not use the modifications in Table 5.2. A major degradation appears in their validation losses supporting the evidence that removal of GNN features degrades ChainNet.	122
5.14	(a): An example of different objective function reductions across 5 independent trials. (b): Mean relative loss reduction by ChainNet and baseline method in two groups of experiments. (c)-(d): The change process of mean loss probability and mean relative loss reduction over the fixed time frame. Dashed lines represent estimated results by ChainNet, while solid red lines represent simulated results of ChainNet decisions. The curves show a steep improvement of ChainNet in a short amount of time.	126
5.15	(a)-(b): The change process of mean loss probability and mean relative loss reduction over the same number of steps. The curves indicate that ChainNet is accurate in capturing trends of simulation-based search.	127

Chapter 1

Introduction

1.1 Motivation and Research Objectives

Distributed systems have experienced significant growth and become ubiquitous in recent years [TVS17, VWK⁺20]. This evolution has been driven by the need for scalable, dependable, and efficient solutions in numerous fields such as business processes, cloud services, and edge computing. A distributed system typically involves a set of resources that work collaboratively to complete a common task. During the design phase, the same functional requirements can be achieved by different design solutions. However, they may vary significantly when evaluated from a non-functional perspective [CDMI11]. *Performance* serves as a core non-functional attribute, which involves the quantitative characteristics of a system, such as response time and throughput. System designers typically face the question of comparing design alternatives and choosing the ones that meet the quality of service (QoS) requirement. As distributed systems continue to increase in complexity and scale, ensuring their performance has become an ever more challenging task.

Performance modeling is the process of representing system behavior using specific notations and quantitatively evaluating performance characteristics, making it crucial in the early design phase of a system. Simulation-based and analytical model-based approaches are the two primary techniques in this domain [BGDMT06]. The simulation approach allows for the creation

of a detailed abstract representation for the system, yielding highly accurate results. However, this technique is typically burdensome to use within an optimization framework for developing complex systems, where many thousands of design alternatives may need to be simulated to find an optimal decision. In contrast, the analytical approaches derive the performance using a set of mathematical equations. The major advantage of the analytical approaches over simulation is that the solutions can be obtained quickly, which enables system designers to compare different design alternatives and assess system behavior under various load conditions.

The most widely used analytical approach in performance modeling is *queueing analysis*, a probabilistic method to analyze systems with job queues and servers [TBBR13]. Queueing analysis uses the stochastic process as a mathematical framework in which it derives formulas that characterize the performance, following the theory that the quantitative behavior of a real system during a specified period of time can be described by the probability distributions of a stochastic process [DB78]. This method relies on a particular class of performance models known as *queueing networks* (QNs), which provide a structure-oriented description of the system using a directed graph and a set of parameters related to workloads and service demands [BGDMT06]. Applying queueing analysis is cost-effective and offers theoretical insights into quantitative behavior of the system across distinct regions of the parameter space.

Unfortunately, while queueing networks are inexpensive to solve and can be used to aid system design, they typically need strict assumptions to remain tractable. The assumptions underlying queueing networks, such as those related to time distributions and scheduling disciplines, may not hold in real-world distributed systems which are characterized by a high degree of complexity and interdependency [CDK05]. Moreover, as modern distributed systems evolve, they may introduce new types of workloads or changes in service architectures that deviate significantly from the scope of traditional queueing analysis. Queueing networks are generally categorized into tractable and unsolved classes [CS78]. While most analytical approaches focus on the tractable category, particularly on *product-form* queueing networks, more realistic queueing networks are usually unsolved. For the unsolved category, most practices have to rely on simulation or approximation techniques, which burden them with high time costs or limited accuracy. Therefore, our aim is to bridge this gap by developing a general methodology capable

of providing accurate predictions across a wide range of queueing network models.

We identify two main challenges in traditional queueing analysis. The first challenge is response time distribution estimation. While mean values provide a good summary, they might be insufficient for characterizing key performance characteristics, such as response time. System designers often need information on the variability of response times or the percentiles of distributions [Har93]. Although computing mean performance values has been extensively studied, little progress has been made in obtaining response time densities. This has proven problematic for traditional queueing analysis [CS78], with successful solutions only limited to a small set of simple networks with specific topological structures, such as a tandem pair of single-server service centers with a few jobs. The root cause of this difficulty is state-space explosion. For example, in the analysis of closed Markovian queueing networks, the traditional approach involves tagging a job and enumerating all possible states, transforming the problem into finding the distribution of time to absorption in a continuous-time Markov chain, which can be intractable for large chains [GATW11]. Closed-form solutions for response time densities rarely exist, even a slight increase in network complexity can lead to a rapid growth in the number of states and may exhaust the computation [TRF03].

The second challenge in queueing analysis is obtaining accurate performance predictions for non-product-form queueing networks. Although many studies address product-form queueing networks, most real-world queueing problems result in non-product-form networks, which often remain unsolved. For instance, a network of service centers with finite-buffer queues, which is common in resource-constrained edge environments [DRA21], often deviates from the infinite capacity assumptions typically made in analytical methods due to the difficulty of characterizing correlations between queues. Examples of other non-product-form networks include those with first-come-first-served service centers where service times depend on job classes, mechanisms that block, suspend, or drop jobs, and so on [BGDMT06]. For non-product-form queueing networks, even estimating mean performance values with analytical methods is challenging. If these networks are Markovian, performance values can be obtained through the numerical solutions of the underlying continuous-time Markov chains. However, this approach is severely limited by the space and time complexities that grow exponentially with the number of com-

ponents of the queueing network [BPI03]. Approximate solutions are therefore developed, but they are often tailored to a narrow set of problems and rely on strict assumptions regarding the number of job classes, scheduling disciplines, and other factors [CHW75], limiting their general applicability.

Nowadays, *machine learning* (ML) approaches are expanding at an accelerating pace and solving complex problems in various domains [JM15, Alp21, ZZL⁺23], the effect of machine learning has been widely felt across most sectors. Machine learning is a subfield of artificial intelligence that uses learning algorithms to train and make predictions *without explicit programming*. Unlike analytical approaches, machine learning models can learn directly from data, identifying patterns and relationships that might be difficult to capture through conventional means. The limitations imposed by certain assumptions required to make queueing networks tractable or solvable do not apply to machine learning approaches. Therefore, there is a generalized belief across the community that machine learning has great potential to advance performance modeling by overcoming the barriers encountered in traditional queueing analysis.

However, the machine learning approach comes with its own challenge when applied to queueing networks. Queueing networks can exhibit various topologies because their service centers may be connected by flexible routes of jobs. Although analytical approaches have strict assumptions, they impose fewer constraints on topology. In contrast, most machine learning approaches require fixed input shapes, making it challenging to develop a learning-based solver that suits queueing networks with unseen topologies. In fact, most literature uses their learning algorithms to study a relatively small queueing network with a fixed topology [NOW20, NPD21, BKSS22, CH23]. However, it would be impractical to rely on a large collection of learning-based solvers, each tailored to a specific queueing network topology, for system design. The scarcity of research on learning algorithms for queueing networks with flexible topologies locks the potential for the general use of learning-based solvers by system designers.

Given the strengths and weaknesses of both queueing analysis and machine learning approaches, in this thesis we explore combining these two approaches to create a more comprehensive performance modeling framework. The integration of queueing theory with machine learning offers

the potential to leverage the best aspects of both methodologies, resulting in models that are both accurate and can be applied to a broad collection of queueing networks. In such a model, queueing analysis can provide a structural framework, offering insights into the fundamental behavior of system components and making learning-based solver more applicable to different queueing network topologies, while machine learning can be used to relax assumption constraints through observing data, capturing the complexities and interdependencies that may not be fully addressed by a purely analytical approach.

1.2 Thesis Organization and Contributions

The structure of this thesis is organized as follows.

- Chapter 1 outlines the motivation, identifies the research gaps, and highlights the main contributions of this thesis.
- Chapter 2 introduces the background knowledge, including the mathematical basics of performance modeling and relevant machine learning techniques, to aid understanding.
- Chapter 3 presents a solution for estimating response time densities in layered queueing networks, which corresponds to Contribution 1.
- Chapter 4 develops a graph learning-based solver for the performance evaluation of closed queueing networks, which corresponds to Contribution 2.
- Chapter 5 proposes a surrogate model for solving multiclass loss networks to aid loss-aware Edge AI service deployment, which corresponds to Contribution 3.
- Chapter 6 concludes the thesis and presents potential directions for future research.

The focuses of Chapters 3, 4, and 5 are briefly illustrated in Figure 1.1.

The contributions of this thesis are summarized as follows.

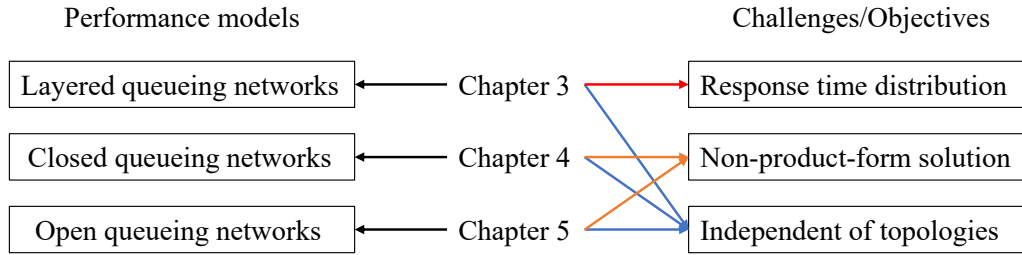


Figure 1.1: Main developments and corresponding thesis chapters.

- The thesis addresses the challenge of estimating response time densities in complex performance models of distributed systems by providing a novel solution that combines neural density estimators with a Markovian analytical framework.** We use layered queueing networks (LQNs) [WF02] as the reference abstraction of distributed systems, and propose a density-passing framework. In this framework, we recursively predict response time distributions in basic components of a layered queueing network by neural density estimators. We describe these response time distributions as a mixture of density functions for which we learn the parameters through a mixture density network (MDN). In order to characterize the density of the end-to-end response time, our density-passing framework recursively propagates MDN predictions across layers using PH distributions and performs repeated distribution refitting. This solution is proved to be effective in accurately estimating response time densities in layered queueing networks with different topologies.
- The thesis is the first to demonstrate that the barrier preventing learning-based models from serving as a general-purpose solver for queueing networks can be eliminated through a proper use of graph neural networks (GNNs).** Inspired by the resource interdependencies in closed queueing networks, we develop a graph learning-based solver that expresses a closed queueing network as a graph comprising global nodes for job classes and local nodes for service centers, iteratively exchanges messages between them and accurately predicts main performance measures per job class at each service center. The key advantage of this solver is its ability to operate independently of the number of job classes and topologies, enabling it to solve a broader

range of closed multiclass queueing networks. We also demonstrate the development of enhanced learning strategies and their effects on improving prediction accuracy. Additionally, our customized solver shows better performance compared to the off-the-shelf GNN blocks. These findings could open a new horizon for developing general solutions for non-product-form queueing networks that pure analytical methods cannot address.

- **The thesis proposes a graph leaning-based solver tailored to finite-capacity queues and job loss abstracted from distributed edge AI deployment, tackling the difficulty of analyzing multiclass loss systems and enabling loss-aware system design.** We use insights from analyzing open queueing networks to meticulously customize a graph learning-based solver called ChainNet, which learns interdependencies along chains and within components of a chain to predict the chain loss. ChainNet serves as a surrogate mathematical model to assess the reliability of deployment alternatives and guide the search for an optimal deployment plan. Extensive results show that ChainNet substantially improves loss prediction accuracy compared to established GNN models. In addition, we propose a detailed design that equips ChainNet with out-of-distribution (OOD) generalization ability, enabling accurate predictions when encountering increasingly complex topologies and dependencies. Moreover, we show that ChainNet provides significantly more dependable deployment decisions under a fixed time budget compared to simulation-based search. ChainNet has the potential to act as a general framework for applying GNNs to model distributed deep neural network (DNN) deployments.

1.3 Publications

The work in this PhD thesis has led to the following publications (sorted by date):

[NRC24] Zifeng Niu, Manuel Roveri, Giuliano Casale. **ChainNet: A Customized Graph Neural Network Model for Loss-aware Edge AI Service Deployment**. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2024.

This work proposes ChainNet, a customized GNN model developed to assess the reliability

of edge AI deployments. ChainNet demonstrates high accuracy in predicting data loss and exhibits strong OOD generalization capability. It effectively serves as a surrogate model to guide deployment optimization. A detailed description of this work is provided in Chapter 5.

[Niu24] Zifeng Niu. **Graph Learning based Performance Analysis for Queueing Networks**. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) Performance Evaluation Review, 2024. *Bronze medal in ACM sigmetrics student research competition 2023*.

This work leverages a graph learning-based method to evaluate the system performance of single-class tandem queueing networks with blocking. The experiments suggest that using GNNs as performance predictors for queueing networks can deliver acceptable accuracy and high time efficiency. The model used in this work serves as a baseline method in Chapter 4.

[NC24] Zifeng Niu, Giuliano Casale. **Neural Density Estimation of Response Times in Layered Software Systems**. IEEE Transactions on Software Engineering (TSE), 2024.

This work extends [NC21] mainly by proposing PH modeling for characterizing service requests between layers and the overall service demand within each layer. This leads to an end-to-end PH analytical framework that facilitates the tractability of response time probability distributions across the entire system. A detailed description of this work is provided in Chapter 3.

[NC21] Zifeng Niu, Giuliano Casale. **A Mixture Density Network Approach to Predicting Response Times in Layered Systems**. IEEE International Symposium On Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2021.

This work proposes an approach to estimate response time densities in LQN models. The method decomposes an LQN into a set of submodels, using MDNs to predict response time distributions and PH distributions to propagate densities, followed by convolutions to obtain the approximation. Experimental results verify the effectiveness of the approach.

Besides, the following are joint works by our research group, Quality of Service Research Lab (QORE).

[CGNZ23] Giuliano Casale, Yicheng Gao, Zifeng Niu, Lulai Zhu. **LN: A Flexible Algorithmic Framework for Layered Queueing Network Analysis**. ACM Transactions on Modeling and Computer Simulation (TOMACS), 34 (3), 1-26, 2023.

[CGNZ22] Giuliano Casale, Yicheng Gao, Zifeng Niu, Lulai Zhu. **LN: A Meta-solver for Layered Queueing Network Analysis**. International Conference on Quantitative Evaluation of Systems (QEST), 232-254, 2022.

These works provide an overview of LN, a meta-solver included in the LINE software package for analyzing LQN models. The LN solver offers users different types of advanced solution paradigms to use. I mainly contributed to the introduction of layering strategy and the case study on multi-solver-based layer analysis, which is related to the background of Chapter 3 in this thesis.

Chapter 2

Background

2.1 Performance Modeling

Performance modeling typically refers to the quantitative evaluation of system behavior by specific modeling notations, which is often integrated with system design to ensure that a system meets specified quality of service (QoS) requirements. The primary motivation for performance modeling is to address performance concerns during system development [CDMI11]. Figure 2.1 illustrates how performance modeling is used in the system design process.

The process begins with abstraction, where a formal description of the planned system is created using a high-level modeling notations, such as queueing networks (QNs). The second step involves the computation of performance measures by applying appropriate solution methods, which can be mainly classified into analytical methods and discrete-event simulations. The third step uses the obtained performance measures to verify whether the design meets its performance requirements. If the feedback is positive, system development can proceed to the next phase, otherwise, the planned system must be redesigned to avoid potential violations of performance requirements [BGDMT06]. This evaluation cycle continues until an enhanced design with the desired performance is achieved. As demonstrated, accurate performance evaluation plays a key role in making the entire process a virtuous cycle.

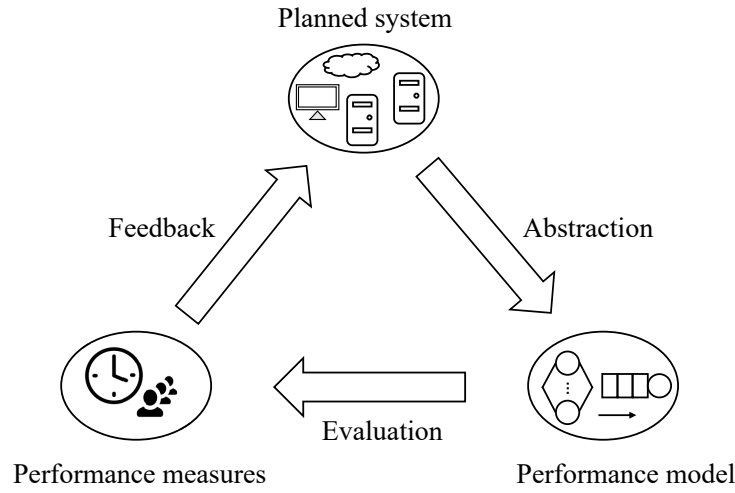


Figure 2.1: The use of performance modeling in the system design process.

2.2 Basics of Queueing Theory

2.2.1 Exponential Distribution and the Poisson Process

The *exponential distribution* acts as the most fundamental continuous probability distribution in queueing theory. A random variable X is said to have an exponential distribution with rate μ , denoted as $X \sim \text{Exp}(\mu)$, if

$$f(x) = \begin{cases} \mu e^{-\mu x}, & x \geq 0 \\ 0, & x < 0 \end{cases}. \quad (2.1)$$

It can be observed that the probability density function decreases by a constant factor $e^{-\mu}$ for each unit increase in $x \geq 0$.

The mean of the exponential distribution is

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx = \frac{1}{\mu}. \quad (2.2)$$

The cumulative distribution function is

$$F(x) = P\{X \leq x\} = \begin{cases} 1 - e^{-\mu x}, & x \geq 0 \\ 0, & x < 0 \end{cases}. \quad (2.3)$$

An important characteristic of the exponential distribution is the *memoryless property*, which describes a situation that the past history of a random variable has no bearing on its future behavior

$$P\{X > t + \Delta t \mid X > t\} = P\{X > \Delta t\}, \quad \forall t, \Delta t \geq 0. \quad (2.4)$$

This equation means that the probability of X exceeding $t + \Delta t$, given that X has already exceeded t , is the same as the probability of X exceeding Δt . In other words, the time already spent waiting for an event does not affect the length of the remaining wait. The exponential distribution is memoryless because

$$P\{X > t + \Delta t \mid X > t\} = \frac{P\{X > t + \Delta t\}}{P\{X > t\}} = \frac{e^{-\mu(t+\Delta t)}}{e^{-\mu t}} = e^{-\mu \Delta t} = P\{X > \Delta t\}. \quad (2.5)$$

In stochastic modeling, the interarrival times of jobs and their service times are often assumed to be exponentially distributed, which allows for the use of Markov chains in system analysis.

The *Poisson distribution* is a discrete probability distribution that is widely used in queueing theory. We say that a random variable X has a Poisson distribution with parameter λ , denoted as $X \sim \text{Poisson}(\lambda)$, if

$$P\{X = n\} = \frac{e^{-\lambda} \lambda^n}{n!}, \quad \text{where } n = 0, 1, 2, \dots; \lambda > 0. \quad (2.6)$$

The mean of the Poisson distribution is

$$E[X] = \sum_{n=0}^{\infty} n P\{X = n\} = \lambda. \quad (2.7)$$

Let $N(t)$ denote the number of events that occur in a time interval $(0, t]$. A *Poisson process* with rate λ refers to a sequence of events such that the times between successive events (i.e., interarrival times) are independent and identically distributed exponential random variables with rate λ and $N(0) = 0$ [HB13].

For a Poisson process with rate λ , the number of events occurring within a time interval is only

determined by the length of the interval, regardless of its starting point. That is

$$P(N(t + \Delta t) - N(t) = n) = \frac{e^{-\lambda\Delta t}(\lambda\Delta t)^n}{n!}, \quad \forall t, \Delta t \geq 0. \quad (2.8)$$

The Poisson process is frequently used to approximate the arrival process in queueing analysis.

2.2.2 Markov Chains and Phase-Type Distributions

A *stochastic process* is defined as a sequence of random variables $\{X(t) \mid t \in T\}$. The parameter t typically represents time, and $X(t)$ is called the *state* at time t . All possible states make up the state space of the process. A process is called a chain if its state space is discrete. The index set T can be discrete or continuous.

A *discrete-time Markov chain* (DTMC) is a stochastic process $\{X(t) \mid t = 0, 1, 2, \dots\}$ that satisfies the *Markovian property*

$$P\{X(t+1) = j \mid X(t) = i, \dots, X(1) = i_1, X(0) = i_0\} = P\{X(t+1) = j \mid X(t) = i\}, \quad (2.9)$$

where $i_0, i_1, \dots, i, j$ denote states.

The Markovian property states that the future state of a process is only dependent on its present state, and is independent of past states.

The $P\{X(t+1) = j \mid X(t) = i\}$ in (2.9) is called the *transition probability* and is denoted by P_{ij} . The *transition probability matrix* of a DTMC, denoted by $\mathbf{P}$, is constructed by placing P_{ij} in the i -th row and j -th column. The elements of $\mathbf{P}$ satisfy $\sum_j P_{ij} = 1$ and $0 \leq P_{ij} \leq 1$, $\forall i$.

The P_{ij}^n represents the probability of being in state j after n -step transitions, given that we start in state i . Let

$$\pi_j = \lim_{n \rightarrow \infty} P_{ij}^n. \quad (2.10)$$

π_j is the *limiting probability* that the chain is in state j infinitely far into the future. It is

noteworthy that π_j is not dependent on the starting state i . For an S -state DTMC with states $1, \dots, S$,

$$\boldsymbol{\pi} = (\pi_1, \pi_2, \dots, \pi_S), \quad \text{where} \quad \sum_{i=1}^S \pi_i = 1, \quad (2.11)$$

is called the *limiting distribution* of the chain. Let $\boldsymbol{\pi}$ denote a probability distribution, this distribution is said to be *stationary* for the chain if

$$\boldsymbol{\pi} \mathbf{P} = \boldsymbol{\pi} \quad \text{and} \quad \sum_{i=1}^S \pi_i = 1. \quad (2.12)$$

They are called *stationary equations*. If the present probability distribution across the states is $\boldsymbol{\pi}$, then after one transition, the probability of being in each state will still be $\boldsymbol{\pi}$. This is the reason for referring to $\boldsymbol{\pi}$ as the stationary distribution. An alternative way of expressing the stationary equations is

$$\sum_{i=1}^S \pi_i P_{ij} = \pi_j, \forall j \quad \text{and} \quad \sum_{i=1}^S \pi_i = 1. \quad (2.13)$$

Assuming that the limiting distribution exists, it is the unique stationary distribution.

We know that $\pi_i = \pi_i \sum_j P_{ij} = \sum_j \pi_i P_{ij}$. From the stationary equations, we have $\pi_i = \sum_j \pi_j P_{ji}$. Therefore, we have

$$\sum_j \pi_i P_{ij} = \sum_j \pi_j P_{ji}, \quad (2.14)$$

which indicates that the total transition rate out of state i is equal to the total transition rate into state i . By removing $\pi_i P_{ii}$ (i.e., self-loops) from both sides of (2.14), this equation can be expressed as

$$\sum_{j \neq i} \pi_i P_{ij} = \sum_{j \neq i} \pi_j P_{ji}, \quad (2.15)$$

The set of equations (2.15) across all states i is called the *balance equations*.

A continuous-time Markov chain (CTMC) is a stochastic process $\{X(t) \mid t \geq 0\}$ that satisfies

the Markovian property

$$P\{X(t_n + \Delta t) = j \mid X(t_n) = i, \dots, X(t_1) = i_1, X(t_0) = i_0\} = P\{X(t_n + \Delta t) = j \mid X(t_n) = i\}, \quad (2.16)$$

where $t_0 \leq t_1 \leq \dots \leq t_n$. In contrast to DTMCs, where transitions happen at discrete time steps, CTMCs allow transitions to happen at any time. The (2.16) implies that the next transition depends only on the present state i , the time the CTMC has spent in i is irrelevant. Therefore, the amount of time before making a new transition is memoryless, which means it is an exponential random variable with some rate.

Let λ_i denote the rate of leaving state i given that the chain is in state i . The set of balance equations for the CTMC is

$$\pi_i \lambda_i = \sum_{j \neq i} \pi_j v_{ji}, \quad (2.17)$$

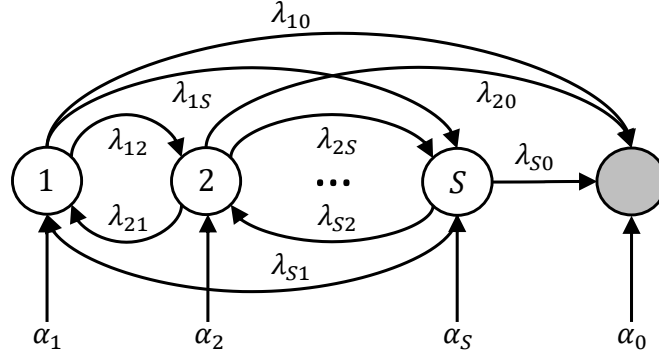
where v_{ji} is the rate of transitions from state j to state i , i.e., $v_{ji} = \lambda_j P_{ji}$. The limiting probability of being in state i , π_i , can be obtained by solving (2.17) with the normalization condition $\sum_i \pi_i = 1$.

Above is a brief introduction to Markov chains, primarily based on [HB13, Ste09], which we recommend for more detailed information.

A *phase-type* (PH) distribution is defined as the distribution of *time to absorption* in a finite CTMC with one absorbing state. Figure 2.2 shows a general $(S + 1)$ -state CTMC with n transient states and an absorbing state. Let $\boldsymbol{\alpha}$ and $\mathbf{T}$ denote the initial probability vector and transition rate matrix among the transient states of the CTMC, that is

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_S),$$

$$\mathbf{T} = \begin{pmatrix} -\sum_{j \neq 1} \lambda_{1j} & \lambda_{12} & \cdots & \lambda_{1S} \\ \lambda_{21} & -\sum_{j \neq 2} \lambda_{2j} & \cdots & \lambda_{2S} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{S1} & \lambda_{S2} & \cdots & -\sum_{j \neq S} \lambda_{Sj} \end{pmatrix}. \quad (2.18)$$

Figure 2.2: An example of a S -phase PH distribution.

The pair $(\boldsymbol{\alpha}, \mathbf{T})$ then determines a PH distribution with the following probability density function

$$f(x) = \boldsymbol{\alpha} e^{\mathbf{T}x} (-\mathbf{T}) \mathbf{1}, \quad \forall x \geq 0. \quad (2.19)$$

The moments of the PH distribution are given by

$$E[X^k] = (-1)^k k! \boldsymbol{\alpha} \mathbf{T}^{-k} \mathbf{1}, \quad \text{where } k = 1, 2, \dots \quad (2.20)$$

In particular, the mean of the distribution is

$$E[X] = -\boldsymbol{\alpha} \mathbf{T}^{-1} \mathbf{1}, \quad (2.21)$$

and the variance is

$$\text{Var}[X] = 2\boldsymbol{\alpha} \mathbf{T}^{-2} \mathbf{1} - (\boldsymbol{\alpha} \mathbf{T}^{-1} \mathbf{1})^2. \quad (2.22)$$

The cumulative density function is

$$F(x) = P\{X \leq x\} = 1 - \boldsymbol{\alpha} e^{\mathbf{T}x} \mathbf{1}, \quad \forall x \geq 0. \quad (2.23)$$

where $\mathbf{1}$ in the equations above denotes a column vector of ones.

Given a probability distribution, multiple PH descriptions could be found. In other words, the same probability distribution may have different $(\boldsymbol{\alpha}, \mathbf{T})$ representations. The series canonical

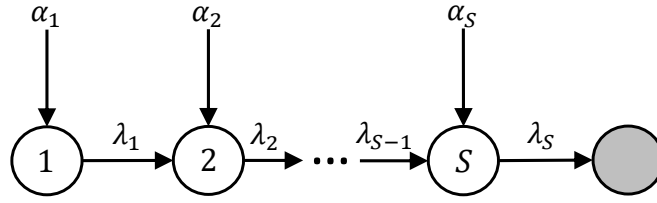


Figure 2.3: The series canonical form of acyclic PH distributions.

form is typically adopted because it has a simple structure with a minimum number of parameters [Cum82]. Figure 2.3 shows the $(S + 1)$ -state CTMC of this form, with the following transition rate matrix

$$\mathbf{T} = \begin{pmatrix} -\lambda_1 & \lambda_1 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\lambda_S \end{pmatrix}. \quad (2.24)$$

For a comprehensive introduction to phase-type distributions, we refer readers to [TB17, He14].

2.2.3 Queueing Networks

Queueing networks are a class of performance models. They are made up of interconnected *service centers*. A service center typically comprises a queue and a finite number of servers. If a service center receives more jobs than servers, then some jobs have to queue and wait for service. A queue may have a finite capacity for holding waiting jobs. New arrivals that cannot be accommodated get dropped. Suppose a center has an infinite number of servers. In this case, no queue will form at the center, which we refer to as a *delay station*. Queueing networks can be categorized into *closed* networks and *open* networks [Whi84]. In a closed queueing network, a fixed number of jobs circulate in the network without any new arrivals or departures. In contrast, in an open queueing network, jobs arrive from external sources, receive services, and then leave the network. Figure 2.4 shows examples of a closed and an open queueing networks. A queueing network may accommodate multiple *job classes*, which may have different workload intensities, service times, and routes in the network.

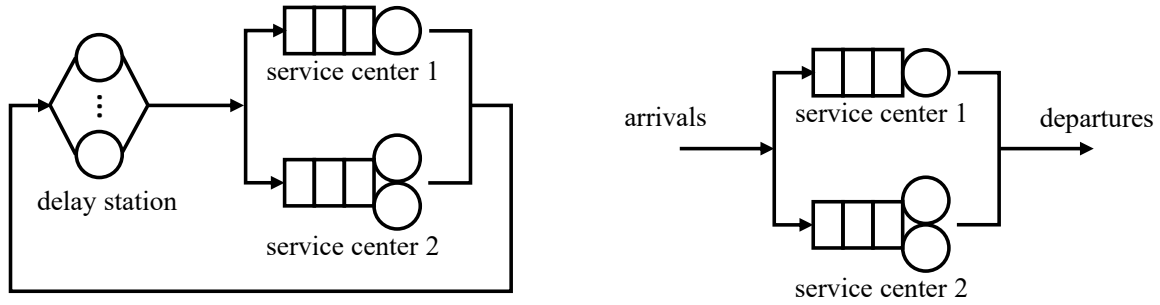


Figure 2.4: Examples of a closed and an open queueing networks.

The *scheduling discipline* of a service center refers to the rule that determines the order in which jobs are served. It selects the next job from the queue when a server becomes available. There are two main types of scheduling disciplines: *non-preemptive* and *preemptive*. Under non-preemptive disciplines, a job runs to completion once it enters a server. In contrast, preemptive disciplines can interrupt a running job during the service, reallocate the server to another job, and then resume the original job later. For example, *First-Come-First-Served* (FCFS) and *Processor-Sharing* (PS) are well-known non-preemptive and preemptive scheduling disciplines, respectively. PS can be regarded as the limit of round-robin scheduling where the quantum size approaches zero, effectively allowing the server to share its time equally and continuously among all jobs.

To evaluate the performance of queueing networks, it is necessary to compute *performance measures* that characterize the quantitative behaviour. For a service center, four performance measures are commonly of interest

- Throughput X : The throughput is the average number of jobs completed per unit time. Suppose we monitor a service center for a period of time, T , during which C job completions occur, then the throughput at this service center is

$$X = \frac{C}{T}. \quad (2.25)$$

- Mean response Time R : The mean response time is the average time that a job spends

waiting in the queue and receiving service in the server

$$R = W + D, \quad (2.26)$$

where W and D denote the mean waiting time and mean service time, respectively.

- Mean queue Length Q : The mean queue length is the average number of jobs in the service center,

$$Q = \sum_n n\pi_n, \quad (2.27)$$

where π_n denote the probability of there being n jobs in the service center.

- Utilization U : The utilization is the proportion of time that each of the servers is busy

$$U = \frac{XD}{m}, \quad (2.28)$$

where m denotes the number of servers at the service center.

For the evaluation of queueing networks, *Little's law* [JC61] acts as the most commonly used equation, which is expressed as

$$Q = XR. \quad (2.29)$$

That states that the mean queue length at a service center is equal to the throughput of jobs multiplied by the mean response time that a job spends at the service center. Little's law has three important advantages [HB13, LZGS84]: (1) It relates three major quantities, which enables us to derive one from the other two. (2) It is independent of service time distributions and scheduling disciplines. (3) In a queueing network, Little's Law can be applied not only to a service center but also at different levels. Figure 2.5 demonstrates this by applying Little's law to a single server, to part of the network, and to the network as a whole.

Many queueing networks can be modeled with CTMCs. Obtaining the probability distribution $\boldsymbol{\pi}$ on the states is central to conventional performance evaluation of queueing networks because the mean values of all other key metrics can be derived from it [BGDMT06]. A general technique

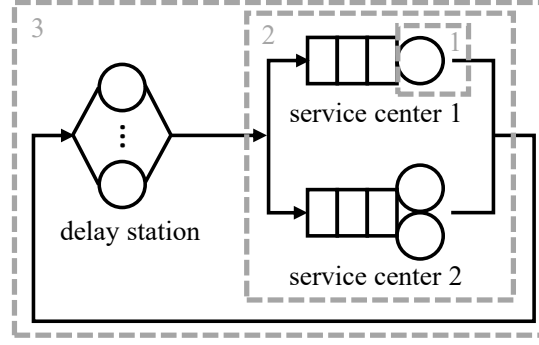


Figure 2.5: Little's law applied at three levels.

for determining π is to solve balance equations (2.17). The balance equation for each state says that the departure rate from a state equals the arrival rate to this state.

For example, let us consider a closed single-class queueing network with K single-server service centers. Jobs are served under FCFS discipline, and service times are exponentially distributed with rates μ_i at each service center. The state of this network can be expressed as $(n_1, \dots, n_K)$, where n_i represents the number of jobs at the i -th service center. Assuming $n_i \geq 1, \forall i$, the balance equation for state $(n_1, \dots, n_K)$ is

$$\pi_{n_1, \dots, n_K} \sum_{i=1}^K \mu_i (1 - R_{ii}) = \sum_{i=1}^K \sum_{j \neq i} \pi_{n_1, \dots, n_{i-1}, \dots, n_{j+1}, \dots, n_K} \mu_j R_{ji}, \quad (2.30)$$

where R_{ij} represents the routing probability from service center i to j . As we can see, the balance equations are complex even for this relatively simple network. For multiclass networks with K service centers and C job classes, the state can be defined as $(\mathbf{n}_1, \dots, \mathbf{n}_K)$, where $\mathbf{n}_i$ is the state of the i -th service center. The $\mathbf{n}_i$ can be defined in multiple ways, such as $(r_i^1, \dots, r_i^{n_i})$ [HB13] or $(n_{i1}, \dots, n_{iC})$ [BGDMT06], where r_i^1 represents the class of the first job at the i -th service center (the one being served), $r_i^{n_i}$ represents the class of the last job at the i -th service center, n_{ir} represents the number of class- r jobs at the i -th service center.

Solving balance equations usually requires resorting to the underlying state space and dealing with a large system of simultaneous equations, which can sometimes be too complex to yield closed-form symbolic solutions. A numerical approximation technique named *matrix-analytic*

method [Neu94] can obtain the solutions for instances of Markov chains by iteration. Still, it suffers from instability caused by singular matrices and high computation costs. Significant progress in the analysis of queueing networks is made by [Jac63, GN67], demonstrating that the solution for $\boldsymbol{\pi}$ in a certain class of networks, known as *product-form* queueing networks, can be expressed as a product of factors corresponding to the state of each service center. This breakthrough offers an immediate closed-form solution for $\boldsymbol{\pi}$ and thus reduces the computational effort considerably. Subsequently, the works [Cha72, CHJT77] show that if each service center in a network possesses the *local balance* property, a product-form solution for $\boldsymbol{\pi}$ exists. The local balance property says that the departure rate from a state due to the departure of a job from the i -th service center is equal to the arrival rate to this state due to an arrival of a job to the i -th service center. For example, the local balance equations for state $(n_1, \dots, n_K)$ studied in (2.30) are expressed as follows

$$\pi_{n_1, \dots, n_K} \mu_i (1 - R_{ii}) = \sum_{j \neq i} \pi_{n_1, \dots, n_{i-1}, \dots, n_{j+1}, \dots, n_K} \mu_j R_{ji}, \quad \forall i. \quad (2.31)$$

The set of product-form queueing networks is extended by [BCMP75]. This work provides detailed assumptions under which product-form solutions exist (e.g., class-independent and exponential service times at an FCFS service center, single-server PS service center). For open queueing networks that meet the assumptions, the solution is simply a product of factors in terms of the input variables. In contrast, for closed queueing networks, the solution involves a normalizing constant $G(\mathbf{N})$

$$\pi_{\mathbf{n}_1, \dots, \mathbf{n}_K} = \frac{1}{G(\mathbf{N})} \prod_{i=1}^K F(\mathbf{n}_i), \quad (2.32)$$

where $\mathbf{n}_i = (n_{i1}, \dots, n_{iC})$, and $G(\mathbf{N})$ is computed by

$$G(\mathbf{N}) = \sum_{\sum_{i=1}^K \mathbf{n}_i = \mathbf{N}} \prod_{i=1}^K F(\mathbf{n}_i), \quad (2.33)$$

where $\mathbf{N}$ is the population vector containing the number of jobs of each class, i.e., $\mathbf{N} = (N_1, \dots, N_C)$. For the computation of $G(\mathbf{N})$, the number of terms needed to add up increases

dramatically in $\mathbf{N}$, K , and C . As a result, obtaining $G(\mathbf{N})$ is typically time-consuming. For a more efficient computation, there has been research into techniques that derive performance measures of closed product-form queueing networks without using normalizing constant. A key approach is *mean value analysis* (MVA) [RL80], it is developed based on *arrival theorem* [LR80, SM81] and Little's law. The arrival theorem says that in a closed product-form queueing network with the population vector $\mathbf{N}$, the distribution of the number of jobs at the service center seen by a class- r arrival to the i -th service center is equal to the distribution of the number of jobs at the service center with one class- r job less in the same network. This theorem relates the computation of the mean response time of class- r jobs at each service center to the mean queue lengths at each service center when there is one fewer class- r job in the network, that is

$$R_{ir}(\mathbf{N}) = f\left(\sum_{s=1}^C Q_{is}(\mathbf{N} - \mathbf{1}_r)\right), \quad (2.34)$$

where $\mathbf{N} - \mathbf{1}_r$ is the intermediate population vector with one class- r job less in the network, i.e., $\mathbf{N} - \mathbf{1}_r = (N_1, \dots, N_r - 1, \dots, N_C)$. Next, the throughput of class- r jobs can be computed by applying Little's law

$$X_r(\mathbf{N}) = \frac{N_r}{\sum_{i=1}^K e_{ir} R_{ir}(\mathbf{N})}. \quad (2.35)$$

Then, the mean queue length of class- r jobs at the i -th service center with $\mathbf{N}$ in the network can be computed

$$Q_{ir}(\mathbf{N}) = X_r(\mathbf{N}) e_{ir} R_{ir}(\mathbf{N}). \quad (2.36)$$

Starting with an empty network where $\mathbf{N} = (0, \dots, 0)$, the population vector is iteratively updated during which we continuously compute $R_{ir}(\mathbf{N})$, $X_r(\mathbf{N})$, and $Q_{ir}(\mathbf{N})$ until the real $\mathbf{N}$ is reached. Consequently, the performance measures of the QN are solved. The detailed steps of mean value analysis [RL80] are summarized in Algorithm 1.

Although mean value analysis eliminates the need to compute the normalizing constant, the memory requirements grow exponentially with the number of job classes in the network [BGDMT06]. To alleviate this problem, a method named *approximate mean value analysis* (AMVA) is pro-

Algorithm 1 Mean value analysis for closed multiclass QNs**Input:** a closed product-form QN with C job classes and K service centers**Output:** $R_{ir}(\mathbf{N})$, $X_r(\mathbf{N})$, $Q_{ir}(\mathbf{N})$ for $i = 1, \dots, K$, $r = 1, \dots, C$

```

1:  $Q_{ir}(\mathbf{0}) = 0$ ; list all  $\mathbf{N}' = (N'_1, \dots, N'_C)$  such that  $0 \leq N'_r \leq N_r$  for  $r = 1, \dots, C$ 
2: for  $n = 1, \dots, \sum_{r=1}^C N_r$  do
3:   for every  $\mathbf{N}' = (N'_1, \dots, N'_C)$  such that  $\sum_{r=1}^C N'_r = n$  do
4:     for  $r = 1, \dots, C$  do
5:       if  $N'_r - 1 \geq 0$  then
6:         for  $i = 1, \dots, K$  do
7:           compute the mean response time of class- $r$  jobs at the  $i$ -th service center,
              $R_{ir}(\mathbf{N}')$ , by (2.34)
8:         end for
9:       end if
10:    end for
11:    for  $r = 1, \dots, C$  do
12:      if  $N'_r - 1 \geq 0$  then
13:        compute the throughput of class- $r$  jobs,  $X_r(\mathbf{N}')$ , by (2.35)
14:        for  $i = 1, \dots, K$  do
15:          compute the mean number of class- $r$  jobs at the  $i$ -th service center,  $Q_{ir}(\mathbf{N}')$ ,
            by (2.36)
16:        end for
17:      end if
18:    end for
19:    complete the iteration for  $\mathbf{N}'$ 
20:  end for
21: end for
22: QN solutions are obtained after the last iteration where  $\mathbf{N}' \equiv \mathbf{N}$ 

```

posed by [Bar79, Sch81]. This method starts with an initial guess for $Q_{ir}(\mathbf{N})$. It then estimates $Q_{is}(\mathbf{N} - \mathbf{1}_r)$ for $s = 1, \dots, C$ based on the initial guess. The value of $Q_{ir}(\mathbf{N})$ is then updated by equations (2.34) to (2.36), which we use to estimate $Q_{is}(\mathbf{N} - \mathbf{1}_r)$ again. We continue iterating until the difference between two consecutive approximations of $Q_{ir}(\mathbf{N})$ falls below a threshold ϵ . The initial guess for $Q_{ir}(\mathbf{N})$ typically assumes class- r jobs are equally distributed over the associated service centers. The work [Bar79] suggests a formula to estimate $Q_{is}(\mathbf{N} - \mathbf{1}_r)$ using $Q_{ir}(\mathbf{N})$

$$Q_{is}(\mathbf{N} - \mathbf{1}_r) = \frac{(\mathbf{N} - \mathbf{1}_r)_s}{N_s} Q_{is}(\mathbf{N}), \quad (2.37)$$

where

$$(\mathbf{N} - \mathbf{1}_r)_s = \begin{cases} N_s, & s \neq r \\ N_s - 1, & s = r \end{cases}. \quad (2.38)$$

Algorithm 2 Approximate mean value analysis for closed multiclass QNs**Input:** a closed product-form QN with C job classes and K service centers**Output:** $R_{ir}(\mathbf{N})$, $X_r(\mathbf{N})$, $Q_{ir}(\mathbf{N})$ for $i = 1, \dots, K$, $r = 1, \dots, C$

```

1: initialize  $Q'_{ir}(\mathbf{N})$ , condition  $\leftarrow true$ 
2: while condition do
3:   for  $r = 1, \dots, C$  do
4:     for  $i = 1, \dots, K$  do
5:       for  $s = 1, \dots, C$  do
6:         estimate  $Q_{is}(\mathbf{N} - \mathbf{1}_r)$  by (2.37)
7:       end for
8:       compute the mean response time of class- $r$  jobs at the  $i$ -th service center,  $R_{ir}(\mathbf{N})$ ,
         by (2.34)
9:     end for
10:  end for
11:  for  $r = 1, \dots, C$  do
12:    compute the throughput of class- $r$  jobs,  $X_r(\mathbf{N})$ , by (2.35)
13:    for  $i = 1, \dots, K$  do
14:      compute the mean number of class- $r$  jobs at the  $i$ -th service center,  $Q_{ir}(\mathbf{N})$ , by
        (2.36)
15:    end for
16:  end for
17:  if  $\max_{i,r} |Q'_{ir}(\mathbf{N}) - Q_{ir}(\mathbf{N})| < \epsilon$  then
18:    condition  $\leftarrow false$ 
19:  end if
20:   $Q'_{ir}(\mathbf{N}) \leftarrow Q_{ir}(\mathbf{N})$ 
21: end while
22: QN solutions are obtained after the changes in  $Q_{ir}(\mathbf{N})$  between two successive iterations
    become sufficiently small

```

The detailed steps are summarized in Algorithm 2. The approximate mean value analysis is further enhanced by [NC81, CN82] where the computation of $Q_{is}(\mathbf{N} - \mathbf{1}_r)$ is modified.

2.2.4 Layered Queueing Networks

Layered queueing networks are a form of extended queueing networks primarily used for modeling complex distributed software systems [GCW19, Woo21, Pet21, WCF22, MBP⁺23]. Figure 2.6 shows an example of LQN that we use to illustrate all of the features considered in the thesis. In LQN models, software resources are called *tasks* and shown as large parallelograms. Tasks are executed by hardware resources called *host processors*, represented by attached circles. When acting as a server, the tasks usually have an FCFS discipline while the processors

are scheduled by PS, a discipline under which the processor runs all jobs simultaneously, and each job experiences an equal fraction of the processor service capacity. The stacked notation makes a task or a processor multiserver, and its multiplicity is indicated in the brace. Replication of servers is not considered. For instance, *AppServer* denotes a task with five worker threads that runs on *App Proc*, which may be a two-CPU multiprocessor.

The different classes of services offered by a task are called *entries* and represented by smaller parallelograms contained by tasks. The execution logic of each entry is depicted as an *activity graph* in which *activities* represent the smallest operation units and are shown as rectangles. Without loss of generality, we assume each entry to have a dedicated activity graph. For example, the execution of *Entry2* in Figure 2.6 is described by an activity graph consisting of six activities, directed arrows, decision/merge nodes (small circles with the + symbol inside), and a loop node (small circle with the * symbol inside). The decision node selects one branch to execute, *A2* with a probability 0.4 and *A3* with a probability 0.6. After running *A3*, *A4* is repeated 2.1 times on average, followed by *A5*. Here, activity graphs that contain fork/join nodes are not considered, we leave this pattern for future work. The *host demand* of each activity, which is the random variable of its processing requirement, is characterized by mean and variance, and the mean value is indicated in the square bracket. The activity graphs are not drawn explicitly if they are only composed of a single activity, in which case, the mean host demand is directly shown in the parent entry.

A key feature of layered queueing models is that activity graphs may include nested *calls* to other services. Calls model remote service requests and are represented by directed arrows going from one activity to an entry of another task. There are two main types of call: *synchronous* and *asynchronous*. The synchronous call is a blocking request during which the sender waits for the reply (dashed arrow), while the asynchronous call is a send-no-reply request, i.e., the execution continues after sending the call. In this paper, we focus on the synchronous call which captures the pattern of standard remote procedure calls (RPC) in most layered software architectures. The amount of calls indicated in parenthesis alongside the request arrow is either *deterministic* or *stochastic*, both cases are considered by our work. Stochastic requests are assumed to be geometrically distributed, and the number in parenthesis is the mean value. In Figure 2.6, the

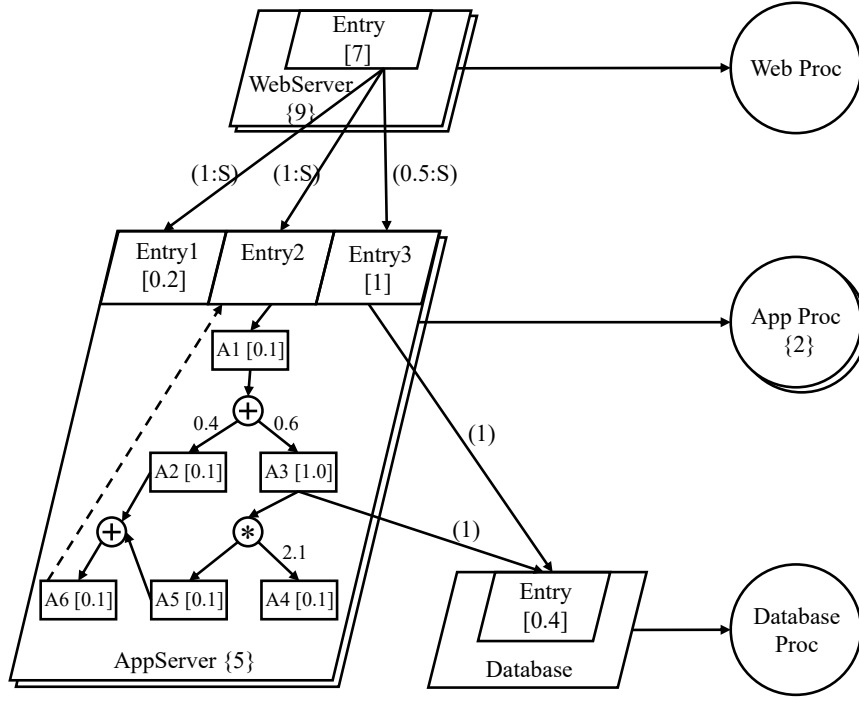


Figure 2.6: An example of an LQN model.

task *WebServer* does not receive any calls. It is the system workload generator named *reference tasks*. We refer to [WF02], [FMW⁺05] for more background on LQNs.

2.3 Machine Learning Approaches

2.3.1 A General Workflow

Machine Learning (ML) aims to make computers learn from the data provided to them, rather than being explicitly programmed with a set of rules. Traditionally, computer programs are written with clear, step-by-step instructions that the computers should take. In ML, however, users create an algorithm that automatically adapts using sample data that describes the desired patterns. The process of automatic adaptation is referred to as *training*. This training process produces a program known as a *trained model*, which comprises the algorithm and a set of parameters it has learned from the training data. The learned parameters, commonly referred to as *weights*, define how the program interprets new data to align with the patterns it has observed. As the model is exposed to more samples, it continues to adapt and improve its

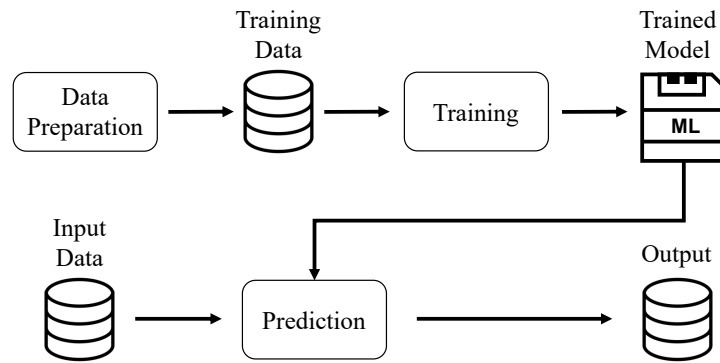


Figure 2.7: Machine learning workflow.

performance without additional explicit programming. It is this adaptability that enables ML to deal with complicated tasks effectively.

Figure 2.7 shows a general workflow of the ML process, which is typically divided into two phases: training and prediction. The training phase involves preparing training data and feeding them into an algorithm that learns and creates a trained model. The prediction phase uses this trained model to make predictions on new input data and produce results.

Neural networks are a significant means of doing ML, which generally fall into two main categories: feed-forward neural networks (FFNNs) and recurrent neural networks (RNNs). FFNNs consist of layers where information moves in a single direction, from the input layer to the output layer. In contrast, RNNs feature feedback loops where the output can be fed back as the input. As a result, information is processed through multiple steps rather than just a single pass as in FFNNs. This makes RNNs particularly suitable for managing sequential data. The training process of a neural network involves calculating the *loss* that quantifies prediction errors, using *backpropagation* to compute the gradients of the loss with respect to each weight, and applying *gradient descent* to adjust the weights to minimize the loss.

2.3.2 Mixture Density Networks

A mixture density network (MDN) is a combination of a mixture model and a fully-connected neural network. We start from introducing the mixture model—a weighted sum of I individual

probability density functions

$$f(y) = \sum_{i=1}^I \pi_i p_i(y|\theta_i), \quad (2.39)$$

where π_i are mixing coefficients such that $0 \leq \pi_i \leq 1$ and $\sum_i \pi_i = 1$, and θ_i are parameters that characterize I kernel functions.

Response time density is a significant performance measure of interest. A mixture model has the potential to express arbitrary probability distributions [Bis94], thus we can use it to model various response time distributions in service systems.

Given the parameter set characterizing the system as an input vector $\mathbf{x}$, we wish to obtain its response time distribution expressed by a mixture model. In other words, our goal is to build a conditional density function of y given $\mathbf{x}$

$$f(y|\mathbf{x}) = \sum_{i=1}^I \pi_i(\mathbf{x}) p_i(y|\theta_i(\mathbf{x})). \quad (2.40)$$

In this case, the weight and density parameters become functions of $\mathbf{x}$: $\pi_i(\mathbf{x})$ and $\theta_i(\mathbf{x})$. The input $\mathbf{x}$ determines a unique mixture model through these two functions. We therefore need to find the functional relationships $\pi_i(\cdot)$ and $\theta_i(\cdot)$ respectively. The central idea of MDNs is to model both functions by a fully-connected neural network. Let O denote the number of observed samples, an MDN is trained by the dataset $\{(\mathbf{x}_j, y_j) \mid 1 \leq j \leq O\}$ and learns the weights of the neural network through maximizing the likelihood

$$\ell = \prod_{j=1}^O f(y_j|\mathbf{x}_j). \quad (2.41)$$

In MDNs, there are different types of kernel functions that can be used. The majority of applications choose Gaussian kernels as mixture components [LL19], [RTL9], [MDG21], [NPD21], a few applications have ever used other kernels such as Gamma or Binomial [DHCI20]. In our study, we consider both Gaussian and Gamma kernels. The Gamma kernel γ is a two-parameter ($\theta = \{\alpha, \beta\}$) probability density function with $\alpha > 0, \beta > 0$. Therefore, a Gamma-MDN needs to model three functional relationships $\alpha_i(\cdot)$, $\beta_i(\cdot)$, and $\pi_i(\cdot)$ to determine the conditional density

function of variable of interest

$$f(y|\mathbf{x}) = \sum_{i=1}^I \pi_i(\mathbf{x}) \gamma_i(y|\alpha_i(\mathbf{x}), \beta_i(\mathbf{x})). \quad (2.42)$$

Figure 2.8 shows the structure of the Gamma-MDN. The output layer consists of three types of nodes that return the function values $\pi_i(\mathbf{x})$, $\alpha_i(\mathbf{x})$, and $\beta_i(\mathbf{x})$. The first type employs softmax as an activation function to turn real values into mixing coefficients that sum to one. For the second and third types, the returned values should be always positive. The following activation function [BG17] is employed to ensure this condition

$$g(x) = 1 + ELU(\xi = 1, x), \quad (2.43)$$

where $ELU(\xi, x)$ is Exponential Linear Unit proposed by [CUH15], and ξ is a positive hyperparameter

$$ELU(\xi, x) = \begin{cases} \xi (e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}. \quad (2.44)$$

This activation function is more stable compared to the exponential activation function because the growth is not fast on the positive side. This helps to avoid instability during the training process [BG17]. The training makes the neural network learn its weights by minimizing our objective loss function, defined as the negative log-likelihood

$$Loss = - \sum_{j=1}^O \log(f(y_j|\mathbf{x}_j)). \quad (2.45)$$

2.3.3 Graph Neural Networks

A graph $G = (V, E)$ is composed of a set of nodes V and a set of edges E , where each edge connects a pair of nodes. Graph-structured data differs from the common data type that mainstream neural networks deal with, such as sequential data (e.g., texts) processed by recurrent neural networks (RNNs) [She20] and grid-structured data (e.g., images) processed

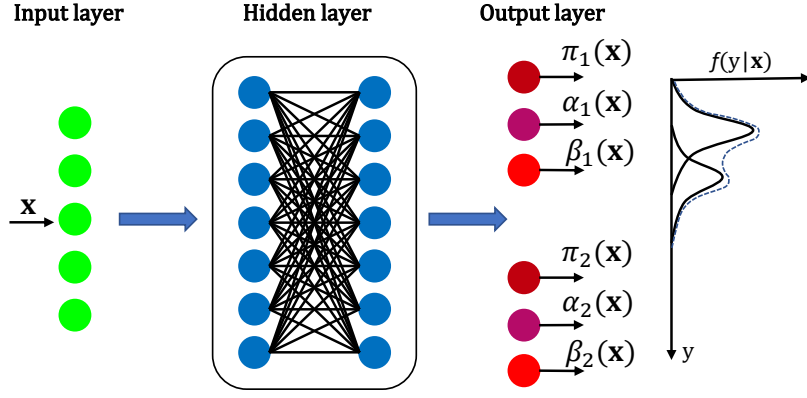


Figure 2.8: The structure of a Gamma-MDN.

Algorithm 3 Neural message-passing framework**Input:** a graph $G = (V, E)$, node features**Output:** $\mathbf{h}_v^{(N)}, v \in V$

- 1: initialize $\mathbf{h}_v^{(0)}, v \in V$, by node features
- 2: **for** $n = 1, \dots, N$ **do**
- 3: **for** $v \in V$ **do**
- 4: node v aggregate the node embeddings from its neighbors
 $\mathbf{m}_v^{(n)} = \text{AGGREGATE}^{(n)}\left(\left\{\mathbf{h}_u^{(n-1)} \mid u \in \mathcal{N}(v)\right\}\right)$
- 5: node v updates its embedding from $\mathbf{h}_v^{(n-1)}$ to $\mathbf{h}_v^{(n)}$
 $\mathbf{h}_v^{(n)} = \text{UPDATE}^{(n)}\left(\mathbf{h}_v^{(n-1)}, \mathbf{m}_v^{(n)}\right)$
- 6: **end for**
- 7: **end for**

by convolutional neural networks (CNNs) [LLY⁺21]. In order to learn on graphs, a specific class of neural networks, known as graph neural networks (GNNs), is proposed to process graph-structured data [SGT⁺08, GSR⁺17].

The input to a GNN model typically comprises a graph and a set of node features. The core idea of GNN approaches is to generate a *node embedding* $\mathbf{h}_v$ for each node $v \in V$, which can be used for downstream prediction tasks [Ham20]. A node embedding is a vector representation of a node that encodes information about its features and surrounding context. To obtain node embeddings $\mathbf{h}_v$, the work [GSR⁺17] presents *neural message passing*, which have become the defining feature of GNNs.

Algorithm 3 presents a general message-passing framework for GNNs. In such a framework, a GNN starts with initial embeddings $\mathbf{h}_v^{(0)}$ and iteratively performs message passing N times to produce $\mathbf{h}_v^{(N)}$ as final node embeddings (lines 1-7). At each iteration n of the GNN, it

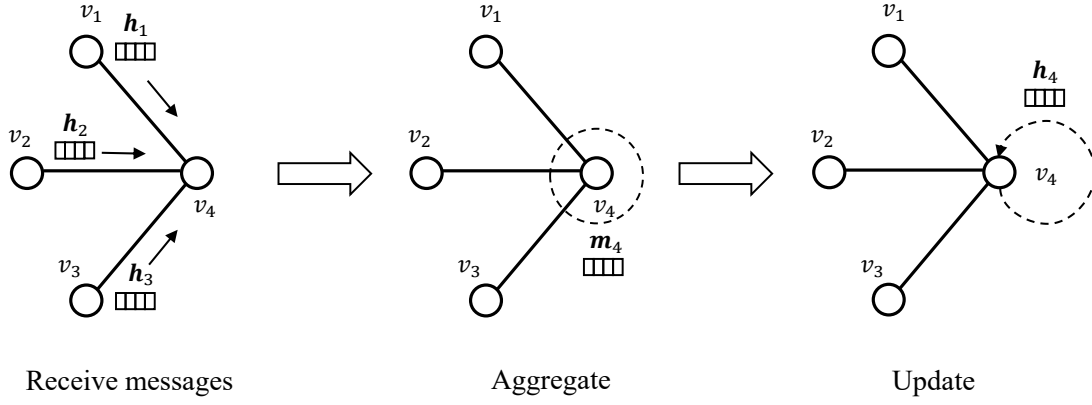


Figure 2.9: One iteration of message passing for a node.

generates an embedding $\mathbf{h}_v^{(n)}$ for each node in the graph. This is achieved by two successive steps: (i) aggregating the node embeddings from neighbors of node v (line 4), known as *message aggregation*, and (ii) updating the embedding of node v using the result of aggregation (line 5), known as *message update*. Two corresponding functions AGGREGATE and UPDATE are allowed to include learnable parameters provided that the functions are differentiable with respect to these parameters. The same function parameters are shared across all nodes at each iteration. They can also be shared across all iterations [BB23].

The aggregation function can take many different forms, and the most basic one is summation [BB23]

$$\text{AGGREGATE}^{(n)}\left(\left\{\mathbf{h}_u^{(n-1)} \mid u \in \mathcal{N}(v)\right\}\right) = \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(n-1)}, \quad (2.46)$$

where $\mathcal{N}(v)$ represent the set of neighbor nodes adjacent to v .

Similarly, there are different possible forms for the update function. A straightforward option would be a single-layer neural network [BB23]

$$\text{UPDATE}^{(n)}\left(\mathbf{h}_v^{(n-1)}, \mathbf{m}_v^{(n)}\right) = \sigma\left(\mathbf{W}_{self}^{(n)} \mathbf{h}_v^{(n-1)} + \mathbf{W}_{neigh}^{(n)} \mathbf{m}_v^{(n)} + \mathbf{b}^{(n)}\right), \quad (2.47)$$

where σ represents a non-linear activation function that has an element-wise operation on a vector argument, and where $\mathbf{W}_{self}^{(n)}$, $\mathbf{W}_{neigh}^{(n)}$, and $\mathbf{b}^{(n)}$ represent learnable weights and biases shared across all nodes.

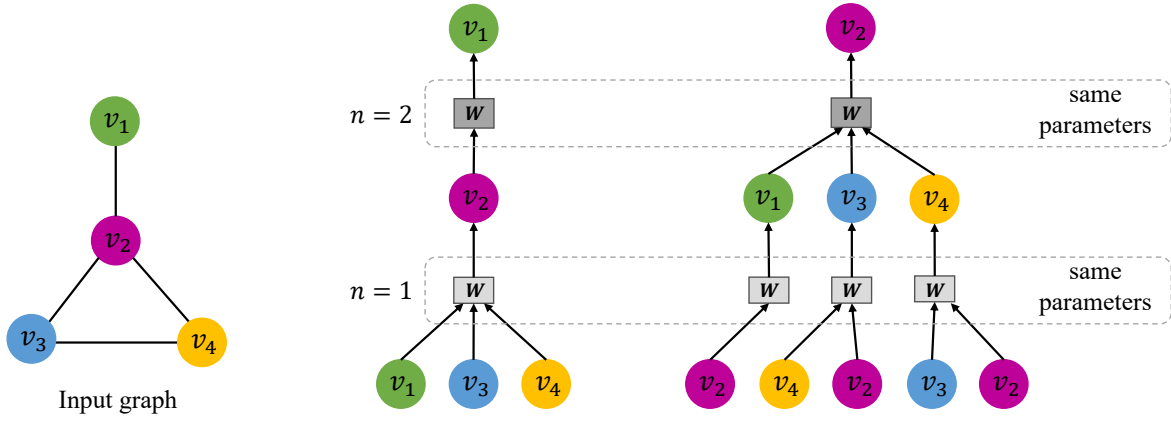


Figure 2.10: Unfolding computation graphs to visualize how a node interacts with the surrounding context through the iterations of message passing.

As shown in Figure 2.9, the GNN message-passing algorithm can be seen as a process where each node receives information from its neighbors, aggregates the information, and uses it to update its own state. This is similar to how RNNs use observations to update their hidden states. Therefore, techniques used to update RNN hidden states can be directly applied to GNNs for updating node embeddings. For example, the gated recurrent unit (GRU) cell [CVMG⁺14], an enhancement of the standard RNN, can be used as an update function

$$\mathbf{h}_v^{(n)} = \text{GRU}\left(\mathbf{h}_v^{(n-1)}, \mathbf{m}_v^{(n)}\right). \quad (2.48)$$

The message-passing GNN defines a unique computation graph for each node based on its neighbors, so every node in the graph has its own neural network architecture [Ham20]. For example, Figure 2.10 displays the computation graphs for nodes v_1 and v_2 in a simple input graph, which are defined by a GNN that involves two message-passing iterations. It can be observed that the nodes v_1 and v_2 use their respective computation graphs to obtain final embeddings, but they share the same parameters of aggregation and update functions at each iteration.

The classic message-passing GNNs involve graph isomorphism networks (GINs) [XHLJ19] and graph attention networks (GATs) [VCC⁺17, BAY21]. GIN models employ a sum-based aggregation and update the node embedding by an MLP. GAT models introduce an attention

mechanism to weigh the importance of each neighbor and update the node embedding with a weighted sum of neighbor embeddings, which involves the following four steps:

To update the embedding of node v at the n -th iteration, the first step is to compute an attention score, e_{vu} , for each neighbor of node v using an attention function

$$e_{vu} = \text{attention}(\mathbf{h}_v^{(n-1)}, \mathbf{h}_u^{(n-1)}, \mathbf{W}^{(n)}, \boldsymbol{\alpha}^{(n)}), \quad u \in \mathcal{N}(v), \quad (2.49)$$

where $\mathbf{W}^{(n)}$ and $\boldsymbol{\alpha}^{(n)}$ represent a learnable weight matrix and a learnable weight vector, respectively. The attention function typically includes a LeakyReLU transformation.

Next, the attention scores are normalized using Softmax

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in \mathcal{N}(v)} \exp(e_{vk})}. \quad (2.50)$$

Then, a weighted sum of node embeddings is performed to obtain an aggregated message

$$\mathbf{m}_v^{(n)} = \sum_{u \in \mathcal{N}(v)} \alpha_{uv} \mathbf{W}^{(n)} \mathbf{h}_u^{(n-1)}. \quad (2.51)$$

Finally, the embedding of node v is updated through a nonlinear transformation

$$\mathbf{h}_v^{(n)} = \sigma(\mathbf{m}_v^{(n)}) = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{uv} \mathbf{W}^{(n)} \mathbf{h}_u^{(n-1)}\right). \quad (2.52)$$

Final node embeddings are generated at the last iteration of the message passing, capturing important structural and feature information within a graph. These embeddings are then fed into a prediction function (i.e., an MLP), which maps node embeddings to predicted values. A loss function is then used to compute the discrepancy between the predicted values and true labels. The GNN keeps minimizing the discrepancy by updating the learnable parameters of its functions. This forms the training pipeline of GNNs, as shown in Figure 2.11.

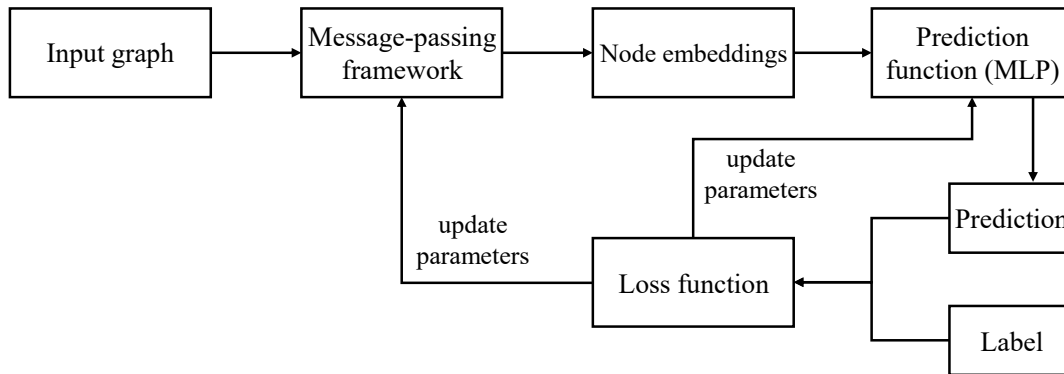


Figure 2.11: GNN training pipeline.

2.4 Related Work

As introduced in Section 1, an important challenge we aim to address is characterizing response time distributions in complex performance models abstracted from distributed systems. In conventional queueing theory, tracking a tagged job is a commonly used way to analyze response time distributions [Har93]. A typical method introduced in [HK02],[Har93] is to derive the Laplace–Stieltjes transform (LST) of passage time for the tagged job in ordinary queueing networks and decondition the start state to obtain the product-form LST of the steady-state response time distribution. However, this method can only be applied when the scheduling discipline is FCFS. In addition, its state-space analysis quickly suffers from state explosion. Fluid approximations have been used successfully to approximate response time distributions in closed networks that consist of delay and processor sharing (PS) queueing stations [ZCP20, RBAC22]. Here, the mean-field fluid approximation based on Kurtz’s theorem [Kur70] is difficult to use in networks with multiclass FCFS stations. All the methods discussed above are limited to Markovian queueing networks. At present, there are few analytical methods that can be applied without specific assumptions on product-form, Markovian properties, and scheduling discipline. Layered queueing networks (LQNs) extend ordinary queueing networks by incorporating nested requests and simultaneous resource possession, which further challenges the conventional response time analysis [FAOW⁺08]. In the analysis of LQNs, gamma distributions are applied to approximate response time distributions [ZW05], but estimation errors can be large in scenarios where clients make a stochastic number of requests to other servers. A

closed-form formula of response times is then derived in [ODF07] aiming at enhancing estimates for such cases. However, it is mainly restricted by two assumptions, (i) stations have a single server, and (ii) service demand distributions of requests are exponential. Another idea arising in the analytical solution for LQNs is to take the sum of variance estimates of requested services [Rei79, FAOW⁺08] to approximate the variance of end-to-end response time. The response time densities estimated by the state-of-the-art using this principle can yet be less accurate in complex LQNs, especially those with multiple job classes and heavy resource contention.

The limitations of conventional analysis have led to an interest in learning-based methods that parameterize the density via neural networks, thus free the estimation from certain assumptions. Among these methods, the mixture density network [Bis94] is well suited for approximating unknown conditional probability distributions. Over the past few years, MDNs have been widely used in distinct areas and proved to be effective in density estimates [DHCI20], [LL19]. In the queueing literature, [RTL19] first uses this technique to predict probability distributions of customer waiting times in an open service system. MDNs are also employed to predict the end-to-end delay distribution in a tandem queueing network by [MDG21] and waiting time distributions in open heterogeneous queues by [NPD21]. These research outcomes highlight the beneficial application of neural density estimators in predicting response time distributions. However, due to the fixed input shapes required by neural networks, these works tend to focus on specific systems. There is currently no research demonstrating how to use machine learning for predicting response time distributions without relying on specific topologies.

We recognize that the lack of topology independence is a significant barrier preventing learning-based models from functioning as general-purpose solvers for queueing networks. Many machine learning techniques have been recently employed to tackle non-product queueing problems that conventional methods are unable to solve, but these applications are similarly constrained by challenges associated with topologies. Multilayer perceptron (MLP) is employed to predict stationary queue-length distributions of M/G/1 queues [BKSS22] and waiting times of GI/G/m queues [NOW20]. Recurrent neural networks (RNNs) are used to encode average dynamics based on mean-field approximations and derive single-class closed queueing networks from system data [GIT20]. Generative adversarial networks (GANs) are utilized to numerically

learn the unknown service time distributions in $G/G/\infty$ queues [OCG⁺21]. A combination of variational autoencoders (VAE) and long short-term memory (LSTM) components is applied to learn the inputs (i.e., known interarrival and service time distributions) of $GI/G/1$ queueing simulations [CH23]. The reinforcement learning (RL) method is employed to control service rates in open single-class tandem queueing networks for QoS assurance [RTL⁺21]. The K-nearest neighbors (KNN) algorithm is used to predict response times for $G/G/1$ queues operating under the shortest remaining processing time scheduling discipline [PSZ21]. However, these learning methods are not designed for graph-structured data, which indicates that they cannot operate on any given queueing networks with arbitrary topologies in an end-to-end fashion. The graph neural network (GNN) emerges as an effective learning method to handle arbitrary graph topology and learn complicated interactions between entities in the graph [SGT⁺08, XHLJ19, WPC⁺20]. This technique has been explored across different research areas such as protein design [SBCV⁺20] and social recommender systems [FML⁺19]. Since queueing networks are graph-structured data, developing a general-purpose performance evaluation method for various complex queueing networks using GNNs is an important question, however, the research on this topic remains unexplored.

GNN serves as a learning-based method that holds significant potential to advance queueing studies, making them promising surrogates for addressing challenging capacity provisioning problems. A key difficulty in this field is optimizing resource allocation in open multiclass queueing networks with job losses, a typical performance model abstracted from modern distributed services, such as the deployment of deep neural networks (DNNs) across distributed devices. The deployment can be optimized with respect to different metrics such as latency and throughput [CYB⁺21, PK22, LLL⁺21]. Due to the resource constraints of available devices, a DNN model sometimes has to be partitioned into fragments and deployed in a distributed manner [HL22]. With job requests arriving continuously, the partitioned fragments operate on different resources and collaborate to make inferences. Available works pursue near-optimal system design based on the given workloads, partitioned fragments, and available resources [BDJ20, CSK21, DRA21, PG22]. They mathematically formulate the placement as an optimization problem, which is then solved by heuristic methods or optimizers like

Gurobi. However, these methods are primarily developed based on the assumption that there is no traffic congestion, which is a simplification without analyzing multiclass loss systems. In complex scenarios with queues and loss, deriving the exact mathematical expression for the objective function is often challenging. The analytical method simplifies loss networks by assuming service centers have no queues, so an arrival job either begins service immediately or is rejected if no servers are available [KM02]. The difficulties conventional methods face in analyzing various multiclass loss systems necessitate the use of graph learning-based surrogate to bridge this gap and enable loss-aware capacity provisioning. Although ordinary GNNs can be used, the model typically requires a fine-grained, customized design that adapts the message-passing mechanism to the problem context. Proper customization stands as a powerful framework to enhance performance and address specific challenges across various domains [SBV20, WWS⁺23, FGPSV⁺23, RRXB23, HWP⁺23].

Chapter 3

Neural Density Estimation of Response Times in Layered Software Systems

3.1 Introduction

During the last few decades, performance has increasingly become an influential factor in software engineering [BDMIS04]. End-to-end response time is a key performance index, but in some situations mean value estimation is insufficient for system designers, who may want to know the quantile statistics or the extent of variability to assess compliance to service level objectives. Queueing networks serve as a common performance model used for software analysis [WFP07]. Although the problem of approximating response time distributions in queueing networks has been studied over many years, it remains a challenging task because conventional analytical methods are inherently limited by their underpinning technical assumptions. Besides, many services today, such as distributed software applications or business processes, feature concurrent services structured according to a layered architecture. Such services are not simple to model using ordinary queueing networks since layering is a form of simultaneous resource possession that does not have the product-form solutions [BCMP75], which makes it more challenging to estimate response time distributions.

In this chapter, we propose a framework to provide accurate density estimates of response times

in software performance models with layered architectures. The framework requires estimating response time distributions in a set of ordinary queueing networks that are constructed from a layered queueing network (LQN). Current analytical methods for response time distributions are not accurate even in a single ordinary queueing network since they are limited by strict assumptions on scheduling discipline, number of servers, and number of job classes. Learning-based methods offer a number of opportunities to overcome these issues and one of the goals of this chapter is to understand how machine learning (ML) can benefit software performance modeling. We utilize Mixture Density Networks (MDNs) to approximate the response time distribution conditional on the known parameters of an ordinary queueing network representing a software layer. MDNs use a mixture model to fit a probability distribution and a fully-connected neural network to map the conditional relation. In addition to MDNs for predictions, we use phase-type (PH) distributions to pass density predictions across layers. The proposed framework can estimate response time densities in complex systems and achieve high accuracy in our experiments. This approach therefore offers a novel contribution to the field of QoS assessment for layered models.

The rest of the chapter is organized as follows: Section 3.2 describes the concepts and related techniques that are used in this chapter. Section 3.3 proposes our methodology for estimating response time densities in layered systems. Section 3.4 evaluates the performance of our MDN analyzers and density estimation framework. Section 3.5 draws the conclusions of this chapter.

3.2 Prerequisites

3.2.1 LQN Layering Strategy

Several techniques exist to analyze an LQN [FAOW⁺08],[WNPM95],[RS95],[FWR06],[Tri12]. The main way is to decompose an overall model into a set of submodels (ordinary queueing networks) that admit a product-form solution. Each submodel is then solved via analytical techniques such as approximate mean value analysis (AMVA) [CN82]. The result iterates among

all submodels until meeting certain convergence criteria, then we can obtain mean performance measures of the LQN [Fra00]. SRVN layering [WNPM95] and MOL layering [RS95] are two ways to construct submodels. In this chapter, we adopt SRVN layering.

The SRVN acronym stands for stochastic rendezvous network, a model proposed in [WNPM95] to capture multi-tier client-server computer systems. An SRVN classifies system entities into three groups: pure clients, active servers, and pure servers. Pure clients only send requests, while pure servers only receive requests. Active servers can send and receive requests. Requests are served with an FCFS discipline and use the RPC paradigm, i.e., calls are synchronous and clients are blocked until the server replies.

While our work addresses LQNs, which are a more expressive model class than SRVNs, the SRVN layering style can also be applied to LQNs where it is also called loose layering. This is a hierarchical decomposition that breaks a multi-tier client-server system into a collection of small submodels, each comprising only a single service center and a set of client tasks. Thus, a submodel represents a hardware or software resource (the service center) together with all of its clients. As we consider only synchronous calls between tasks, any submodel can then be represented by a closed queueing network consisting of a service center, a delay station, modeling think times, and a set of closed classes, representing the client tasks.

3.2.2 Queueing Network Submodels

We now overview the definition of the LQN submodels, which follows ideas from prior work [Fra00]. Each submodel s comprises a service center and a set of client tasks. With a slight abuse of terminology, we shall refer to the service center s meaning the service center within submodel s . We also use the terms layer and submodel interchangeably.

Two kinds of submodels are possible in the assumed class of LQNs: (i) *processor-submodels*, typically representing a hardware resource, where the service center is a host processor; and (ii) *task-submodels*, typically describing software resources, where the service center models a task. As an example, Figure 3.1 shows a submodel decomposition for a slice of an LQN. Submodel I

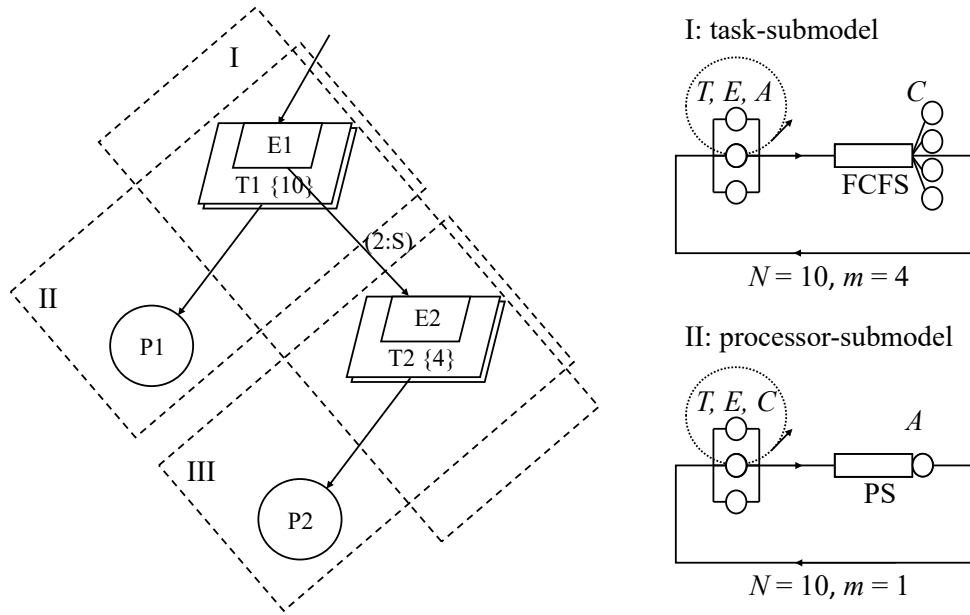


Figure 3.1: Task-submodel and Processor-submodel in an SRVN style decomposition. Submodels I and II are single-class closed queueing networks with 10 jobs, representing workers. The queueing network for submodel III is not shown as it is similar to the one of submodel II.

is a task submodel, with T2 acting as service center and T1 acting as client task. Conversely, submodels II (resp. submodel III) is a processor-submodel, with P1 (resp. P2) acting as service center and T1 (resp. T2) acting as its only client task. Because submodel I depends on the response time at submodel III, and thus is its client, we say that I is an *upper layer* of submodel III, and viceversa III is seen by submodel I as a *lower layer*. Such dependencies define a direct acyclic graph among the layers that can be used to traverse them recursively.

Under the assumed class of LQNs, each submodel may be mapped to a closed multiclass queueing network consisting of a service center and C closed classes, corresponding to the set of client tasks of service center s , each having a population of N_c jobs representing the maximum degree of parallelism for incoming requests from client tasks $c = 1, \dots, C$. The think time Z_c accumulated at a delay station models the mean time between the completion of a service request issued by a worker of client task c and the issue time of the next request from the same worker of c .

The service center in submodel s is also represented by a queue with m servers (workers). The value m matches the multiplicity of the server task s . We admit for the service center only scheduling policies based on either the FCFS or the PS discipline. Sumodels where the service

center is a pure delay may be modeled by using FCFS and setting $m = \sum_c N_c$. Upon visiting the service center, the job receives a host demand having mean D_c and squared coefficient of variation SCV_c . Queueing delays can inflate the host demand resulting in a response time W_c for a job to complete. Due to layering, the distribution of the response time W_c accumulated in a lower layer is then recursively used to parameterize the distribution of the host demand of service center c in the upper layer. However, when the client task is a reference task, rather than a service center, the host demand W_c becomes the end-to-end response time perceived by the system users. Our goal is to characterize the distribution of such end-to-end response time, which will be further examined at the finer granularity of individual entries.

3.2.3 Routing Paths

Response times for jobs in a submodel are accumulated as follows. After first leaving the delay station, the job representing the client task is probabilistically routed among the stations according to a given discrete-time Markov chain. In our models, the job repeatedly visits the delay node and the service center until completion, with the latter corresponding to a reply in its activity graph. In particular, a call to an entry e of service center c in the upper layer may trigger an activity graph that requires service from some service centers s and $s' \neq s$. In the submodel for s , the job will pay visits to the service center to receive the host demand associated to the activities served by s ; instead, any time spent at service center s' will be modeled in submodel s as think time spent at the delay station.

An example of the routing path for a client task is shown in Table 3.1. The table gives the routing probabilities associated with *Entry2* in Figure 2.6 that are used in the lower layer of *App Proc*, where the *AppServer* task acts as a client. Note that the classes in Table 3.1 are named after the LQN entities in Figure 2.6.

In a similar spirit, the small models on the right of Figure 3.1 summarize the station where jobs accumulate time in the four different possible types of job classes we use in activity graphs. In the figure notation, these are: *Task* (T) classes, *Entry* (E) classes, *Activity* (A) classes, and

Call (C) classes. For example, activities in our model always run at the service center of a processor-submodel, which is a host processor.

3.2.4 Modeling Requests to Entries

Our ultimate goal is to characterize the end-to-end response time distribution perceived by the users. A user request is issued by reference tasks and triggers a chain of calls to entries. We thus break the problem of studying end-to-end distributions by characterizing the response time of each entry separately, and later using convolutions of their densities to reconstruct the end-to-end density.

Let us consider the submodel for service center s . To enable such an approach, we study the conditional distribution of the response time random variable W_c for a client task c of s , given knowledge of the entry e of c that triggered the request to service center s . We denote this conditional distribution as W_c^e . If client task c issues the request to s , following invocation of one of the entries e , then we describe this event in submodel s as the job leaving the think state and entering a specific route uniquely associated with entry e . Note that such routing path is simply the route modeling the activity graph of the entry e , similar to the one shown in Table 3.1. Along this routing path for the entry, the job will accumulate into the random variable W_c^e the response times of each visit to the service center. The problem of analyzing entry response time distributions amounts to characterizing the distribution of W_c^e , which follows from the convolution of the response time random variables at these successive visits.

3.2.5 Phase-Type Distributions

Under the assumptions, our LQN models admit a Markovian representation. With this interpretation, an activity graph triggered by a call to entry e may be seen as an absorbing continuous-time Markov chain (CTMC), which models the sequence of response times and think times incurred by the job. The absorbing state corresponds to the traversing the final

Table 3.1: Example of routing path for the activity graph associated to *Entry2* in Figure 2.6. The *App Proc* is a PS station.

Source Station	Destination Station	Source Class	Destination Class	Routing Probability
<i>Delay</i>	<i>Delay</i>	<i>AppServer</i>	<i>Entry2</i>	0.4
<i>Delay</i>	<i>App Proc</i>	<i>Entry2</i>	<i>A1</i>	1.0
<i>App Proc</i>	<i>App Proc</i>	<i>A1</i>	<i>A2</i>	0.4
<i>App Proc</i>	<i>App Proc</i>	<i>A1</i>	<i>A3</i>	0.6
<i>App Proc</i>	<i>App Proc</i>	<i>A2</i>	<i>A6</i>	1.0
<i>App Proc</i>	<i>Delay</i>	<i>A3</i>	<i>Entry</i>	1.0
<i>Delay</i>	<i>App Proc</i>	<i>Entry</i>	<i>A4</i>	1.0
<i>App Proc</i>	<i>App Proc</i>	<i>A4</i>	<i>A4</i>	0.677
<i>App Proc</i>	<i>App Proc</i>	<i>A4</i>	<i>A5</i>	0.323
<i>App Proc</i>	<i>App Proc</i>	<i>A5</i>	<i>A6</i>	1.0
<i>App Proc</i>	<i>Delay</i>	<i>A6</i>	<i>AppServer</i>	1.0

transition modeling the reply of the entry. As PH distributions model times to absorption in CTMC, they can be used to model the resulting response time distribution W_c^e .

The PH distribution is an effective fitting tool in matching two moments. In particular, the PH moment matching technique is always able to return a canonical form $(\boldsymbol{\alpha}, \mathbf{T})$ with the desired mean and variance, provided they are both positive. However, fitting higher moments presents complications due to feasibility bounds that restrict the admissible values of the moments that can be matched [CZS10]. In our work, we therefore just match PHs to the first two moments of response times, leveraging the acyclic PH fitting method proposed in [BHT05].

Lastly, we remark that PH distributions have important closure properties, as studied in [Neu94]; they ensure that certain operations PH random variables produce new random variables that are always PH distributed. These operations include in particular convolutions and mixtures, which are the essential operations we need to model response times in activity graphs.

3.2.6 Client Task and Entry Chains

A client task issues requests to some lower layer s upon invocation of one of its entries. Assume that client task c has K entries, labelled E1 to EK, then K distinct routes will branch out of the delay station in submodel s , one for each entry. To model the frequency of selection of a

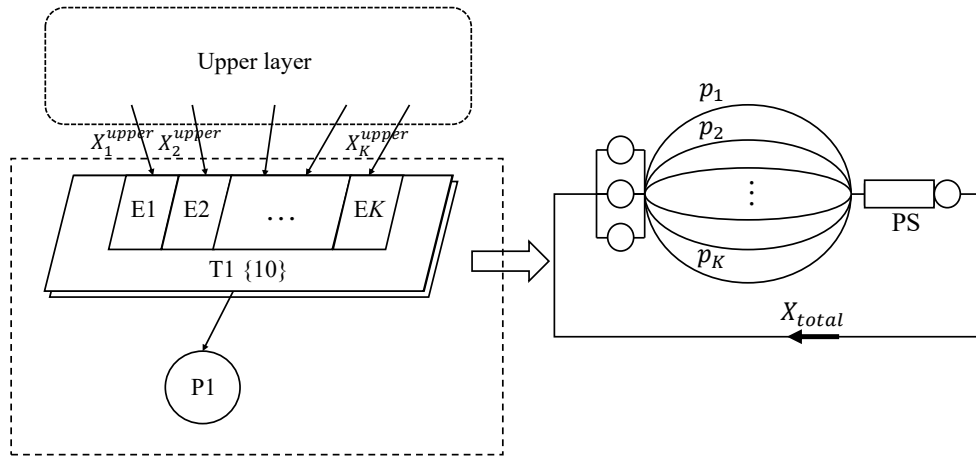


Figure 3.2: A submodel with multiple entries. Each entry is assumed to require a single visit to the processor sharing station modeling host processor P1. The submodel has $N = 10$ jobs, equivalent to the multiplicity of the client task $T1$. The total throughput X_{total} at the lower layer model is equivalent at steady state to the sum of the throughputs X_k^{upper} of each entry $k = 1, \dots, K$.

particular entry of c , each branch is associated with some routing probability p_k , $k = 1, \dots, K$, to be determined by the solver during model solution. An example of this is shown in Figure 3.2. The model has a single client task and $p_k = X_k^{upper} / X_{total}$, where X_{total} is the total throughput (or equivalently, steady-state arrival rate) of the client task and X_k^{upper} is the portion of this throughput due to invocation of entry k .

Under this representation, a job modeling task c in submodel s can over time visit different subpaths representing its K entries. After doing so, it will change class according to the activity graph of the chosen entry, similarly to what shown in the example in Table 3.1. The set of reachable classes for a job modeling client task c is called the *chain* associated to this task. It comprises all the classes reachable by activity graphs of all the entries of c . As a result of our definitions, jobs that describe a client task can switch between subpaths that model entries, but cannot leave the chain that describes c . Thus a chain remains constant over time for every client task c .

The importance of the notion of chain is that the closed queueing network can be solved by considering only aggregated parameters at chain level, such as chain host demands and chain think times. That is, computational methods exist to calculate the performance measures per chain and from then obtain per-class metric in each chain [BB80], [RK75], [RL80], [BGDMT06,

§7.3.6]. From per-class metrics, such as throughputs, response times, utilization, or mean queue-lengths, it is then readily possible to reconstruct corresponding metrics at the level of routing subpaths for each entry.

In light of the last observation, we can leverage chain representations to compactly analyze LQN submodels, without the need for detailed analysis of class switching and thus abstracting out, by means of chain aggregation, unnecessary details of the activity graph structure. In order to instantiate a submodel s it is sufficient to provide a tuple of parameters $\mathbf{x} = (\{N_c, Z_c, D_c, SCV_c\}, m)$, for all chains $c = 1, \dots, C$, each chain being a client task.

Thus, in the next sections, every time we refer to an input vector $\mathbf{x}$, we are taking a chain-level aggregate parameterization of the closed queueing network. In particular, we shall pass these inputs to MDNs, which can therefore be specified without providing as input any activity graph detail, thanks to the chain aggregation technique.

However, the above approach would lead to MDNs that predict only response times of client tasks. To predict conditional entry response times W_c^e , we then build a more detailed model $\mathbf{x} = (\{N_{c,e}, Z_{c,e}, D_{c,e}, SCV_{c,e}\}, m)$ where the entries are mapped to separate chains. This is obtained using approximation, namely a fraction p_e of population N_c is put in the chain for entry e , where p_e is the entry branch selection probability of client task c , which does not present a challenge for methods such as AMVA that can handle fractional populations. Similarly, we set the detailed model parameters as $Z_{c,e} = p_e Z_c$, $D_{c,e} = p_e D_c$, $SCV_{c,e} = p_e SCV_c$. Thus, the new model has one chain for each entry of every client task to the submodel. This detailed model still does not require us to specify the activity graph in the MDN inputs.

3.3 Methodology

3.3.1 Overview

From now on, we assume that the LQN has been solved at equilibrium for mean performance metrics with an appropriate algorithm, such as AMVA [CN82]. Thus, information is assumed

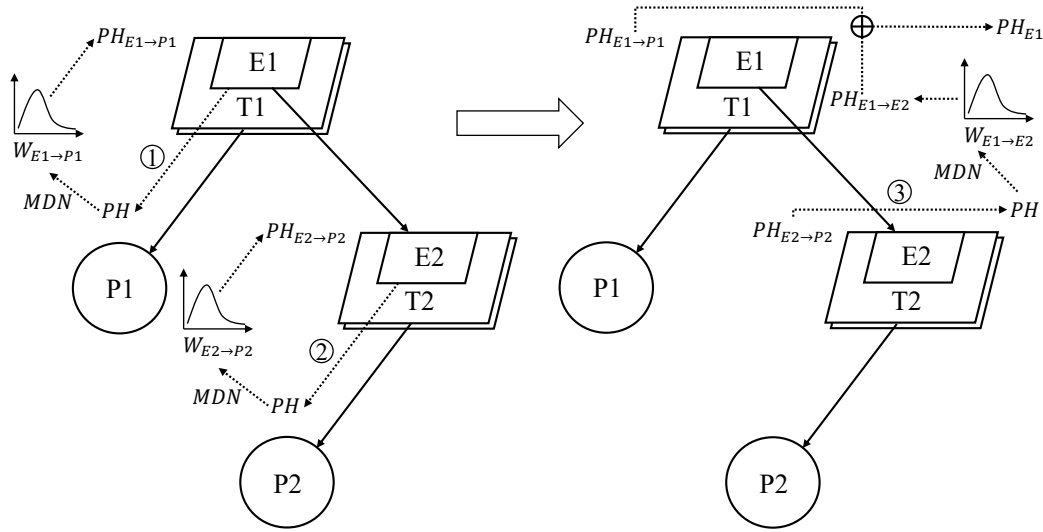


Figure 3.3: Schematic of applying the proposed method on an LQN model. Circles 1, 2, and 3 represent computing host demands of entries using proposed PH operations, and $\oplus$ represents a convolution operation. $W_{A \rightarrow B}$ denotes the random variable for the response time of entry A at B . $PH_{A \rightarrow B}$ denotes the PH representation of $W_{A \rightarrow B}$.

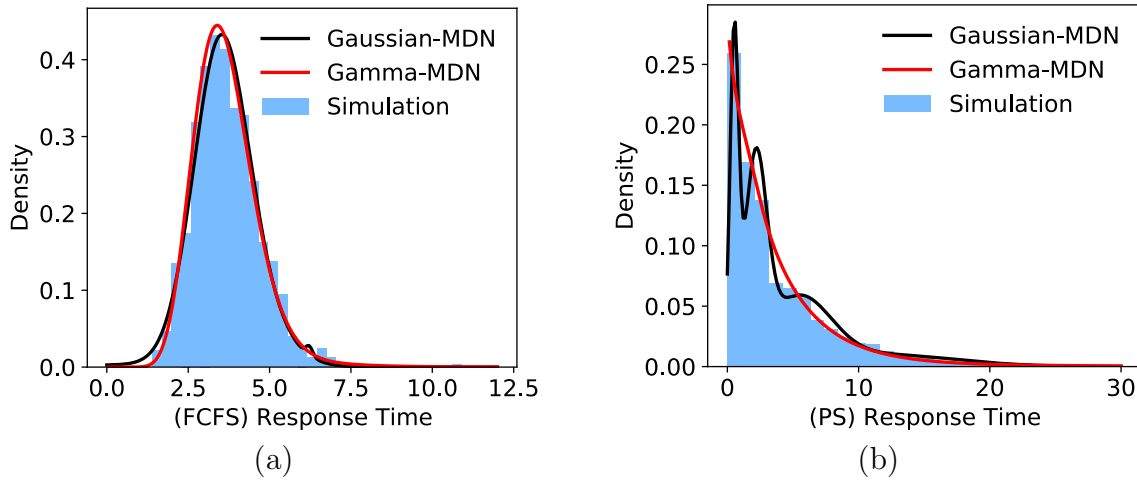


Figure 3.4: A comparison between simulation and estimated conditional density functions of the response time given an input vector $\mathbf{x} = (\{90, 3, 0.6, 1.25\}, 8)$ with (a) FCFS and (b) PS, respectively.

available on the mean response time of every element of the LQN. Note that this means that advanced methods used in mean-value solutions, such as interlocking corrections [Fra00], are applied to the inputs of our approach. Our methodology provides an effective method to augment this mean information with variances and distributions.

As mentioned, to estimate the end-to-end response time distribution E_r of a reference task entry r , we recursively combine the conditional distributions W_c^e visited in the activity graph

of r . A schematic view of the proposed methodology is given in Figure 3.3; the approach recursively processes submodels bottom-up. It generates W_c^e at every submodel until reaching the reference task. Algorithm 4 summarizes the key computational step we apply at each submodel; individual steps are explained next.

As the conditional distributions W_c^e are modeled as PHs, which can be composed, we develop in Section 3.3.4 a method to capture into a single PH representation for the sequence, selection, and loop patterns of the activity graph of an entry e . This is used in Steps 1-3 of the algorithm to estimate the host demand distribution of an entry e of client task c at service center s . The method input requires the PHs for all the activities and calls on this activity graph. The first two moments of the host demand of e are obtained at Step 4. For the think time we use a similar approach, but we retain only times spent at the delay node, instead of at the service center. This last step can be efficiently implemented approximately assuming that the delay station is always insensitive to moments other than the mean, thus the think time $Z_{c,e}$ can just combine the mean values of the times spent at the delay node provided by AMVA as an input to our recursive method.

Given the host demands descriptors $(D_{c,e}, SCV_{c,e})$ and the mean think time $Z_{c,e}$, we can obtain a detailed model $\mathbf{x}$ for which we wish to predict the entry response time distribution W_c^e . This is complex to do for traditional methods under both FCFS and PS scheduling, as we have discussed in Section 2.4. Our methodology addresses the problem using MDNs, which are pre-trained on this task for FCFS or PS service centers, either in single-class or two-class models. For MDNs, we use both Gaussian and Gamma kernels to predict response time distributions in submodels. We find that an MDN with Gamma kernels is more applicable to parameterizing the PS scheduling density model. For instance, Figure 3.4 shows a comparison between one histogram and two density functions obtained from simulation and five-component MDNs respectively. As can be seen from Figure 3.4, both MDNs provide accurate predictions for the FCFS case, but for the PS case, the MDN with Gamma kernels provides a more accurate approximation. The case with more than two classes is reduced to the two-class case as explained in Section 3.3.2. The distribution predicted by the MDN is finally re-fitted to a PH by matching its mean and variance, obtaining the PH for W_c^e .

Algorithm 4 Conditional response time distribution W_c^e **Input:** Activity graph G_e for all entries e of service center c ;PH response distributions of all activity and calls along G_e ;Submodel s .**Output:** PH of W_c^e , for all entries e of c .

- 1: **for** each entry e in client task c **do**
- 2: Temporarily set $p_e = 1$ and $p_h = 0$ for $h \neq 0$, and set the submodel to have a single job, placed in chain c .
- 3: Generate the PH (α_e, T_e) for G_e resolving sequence, selection, and loop patterns among the input PHs and ignoring time spent at the delay node.
- 4: Get from (α_e, T_e) the values of $(D_{c,e}, SCV_{c,e})$ for the host demand of e at s .
- 5: Obtain Z_e as the sum of the mean times spent along G_e at the delay node.
- 6: Restore routing probabilities and submodel population to the original values.
- 7: **end for**
- 8: Predict the response time W_c^e using an MDN with input $\mathbf{x} = (\{N_{c,e}, Z_{c,e}, D_{c,e}, SCV_{c,e}\}, m)$, $\forall e$, where m is the multiplicity of s .

Since the PH for W_c^e also provides the host demand distribution for e at upper layers, the method can be recursively applied. Traversing all the layers in a bottom-up fashion, we end up reaching the reference task and obtaining the response time distribution of E_r , as shown in the right part of Figure 3.3.

3.3.2 Class Aggregation

For a submodel with C client tasks and E_c entries per client task c , $\sum_{i=1}^C E_c$ predictions need to be made by an MDN analyzer, one for each chain in the associated submodel. This can be done by predicting the response time distribution for the first chain and permuting the order of the chains $\sum_{i=1}^C E_c$ times. To reduce the cost of this step, as mentioned we employ a two-class MDN as an approximation. The idea is to focus on a specific chain, and aggregate all the remaining chains into a single one. This idea implicitly leverages SRVN layering, since the approximation error of this aggregation style tends to be low only in small queueing networks with a single service center, such as ones we consider at each layer. In larger models, correlations in demand within a class across multiple service centers would otherwise be lost.

Suppose there are C client tasks and $\sum_{i=1}^C E_c$ entries in a submodel featuring in total of $\sum_{i=1}^C E_c$ chains. We wish to compute the response time distribution of entry e of the client task c in

chain (c, e) . We aggregate the other $\sum_{i=1}^C E_c - 1$ chains into a single chain with population

$$N_{aggr} = \left(\sum_{t=1}^C \sum_{i=1}^{E_c} N_{t,i} \right) - N_{c,e} \quad (3.1)$$

and host demand

$$D_{aggr} = \sum_{t=1}^C \sum_{\substack{i=1 \\ i \neq e \text{ if } t=c}}^{E_c} \frac{X_{t,i}}{X_{total} - X_{c,e}} D_{t,i} \quad (3.2)$$

where $X_{t,i}$ is the throughput of the chain representing entry i of client task t and X_{total} sums the throughputs of all $\sum_{i=1}^C E_c$ chains. The aggregation scheme is inspired by the utilization law and used to aggregate demands also in prior work, e.g., [WCF23]. It weights each aggregated chain (t, i) with a coefficient proportional to the fraction of throughput it generates. The last formula is similarly applied to compute Z_{aggr} , D_{aggr} , and SCV_{aggr} . As a result, we now have an aggregated input vector $\mathbf{x} = (N_{c,e}, Z_{c,e}, D_{c,e}, SCV_{c,e}, N_{aggr}, Z_{aggr}, D_{aggr}, SCV_{aggr}, m)$. This allows us to use an MDN trained on two classes whenever $\sum_{i=1}^C E_c \geq 2$.

3.3.3 Submodel Response Time Predictions

From an input vector $\mathbf{x}$, the MDN output gives parameters of the response time distribution expressed by a mixture model, which are (i) weight parameters $\pi_i(\mathbf{x})$; and (ii) kernel density parameters $\theta_i(\mathbf{x})$. As outlined, we need to convert this prediction into a PH for W_c^e via moment matching. The mixture has in general mean and variance:

$$\mu = \sum_{i=1}^I \pi_i \mu_i, \quad \sigma^2 = \sum_{i=1}^I \pi_i (\sigma_i^2 + \mu_i^2 - \mu^2) \quad (3.3)$$

where π_i , μ_i , and σ_i^2 are kernel parameters returned by this MDN. Using (3.3), we can readily compute by moment matching an acyclic PH that matches these two moments exactly [BHT05]. The resulting PH modeling the response time will also be used to model the host demand of e in the upper layer, thus enabling recursive bottom-up LQN analysis.

To train MDNs that are effective in making response time predictions, we need to use datasets consisting of both input vectors and target outputs. An input vector $\mathbf{x}$ represents a param-

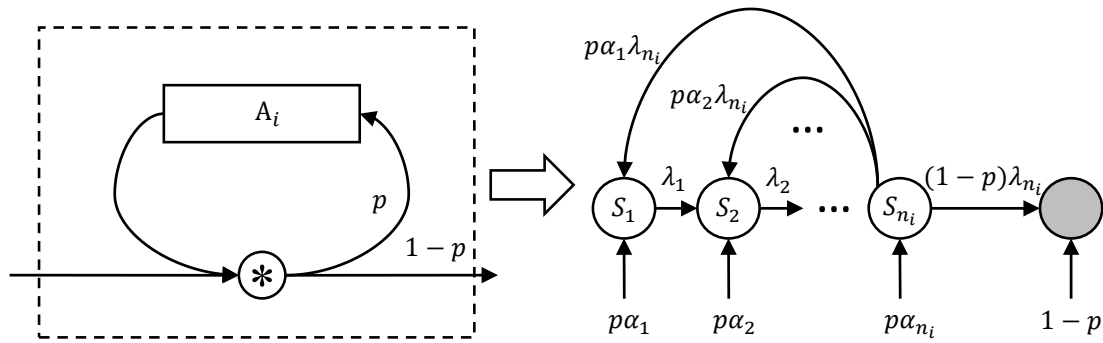


Figure 3.5: Underlying Markov chain of the loop structure.

eterization of the queueing network, thus for each $\mathbf{x}$ we can run a discrete-event simulation of the model and obtain a response time distribution which serves as the target output. To create a dataset for MDN training, we then generate a batch of input vectors by sampling uniformly within the valid parameter ranges of the problem and then obtain their target outputs by discrete-event simulation. In this way, the fitted MDNs possess sufficient generalization capabilities to analyze response times in a broader range of LQN submodel parameterizations.

3.3.4 PH Models of Activity Graphs

To enable the proposed approach, we need a method to model in Step 3 of Algorithm 4 the host demand of an entry in terms of a PH distribution. This requires in particular to cope with the presence of sequence, selection, and loop patterns in the activity graphs. Two aggregation methods are available in prior work to this aim. The approach in [Fra00, p. 183-186] repeatedly aggregates unit structures until obtaining a single activity that can represent the original activity graph; during this process, mean and variance are computed based on summation formulas in [Smi90]. However, the method does not encompass algorithms to process more general graphs and detect patterns. The work in [Isl18, p. 31-33] finds the weight of each activity in the graph and calculates the weighted sum of host demands. In this approach, a method is not available for variance aggregation. Our method differs from these earlier works by relying on PHs, which carry all moments allowed by their degrees of freedom and the associated distribution. Our approach also leverages efficient (sparse) matrix operations to compute PH representations and relevant mixture and convolution operations.

The method proceeds by identifying patterns within the activity graph, and removing them via the formulas presented in the next subsections. For activity graphs that do not include loop patterns, a bottom-up traversal of the graph that applies the proposed transformations will be sufficient to generate a single PH distribution for the whole activity graph. The handling of loop patterns and nested loop patterns is described later in this section.

Note that, because our methodology treats differently host demand accumulated at service centers from think time spent at delay stations, care should be taken in the implementation of the recursive analysis to skip nodes of the activity graph that do not characterize time spent at the considered station.

PH Model of Host Demands

Given an activity graph at a processor-submodel, we first fit the host demand of each activity with the canonical form of acyclic PH distributions. For an activity A_i , suppose the canonical representation $(\boldsymbol{\alpha}_i, \mathbf{T}_i)$ of its host demand has n_i transient states, then

$$\begin{aligned} \boldsymbol{\alpha}_i &= (\alpha_1, \alpha_2, \dots, \alpha_{n_i}), \\ \mathbf{T}_i &= \begin{pmatrix} -\lambda_1 & \lambda_1 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\lambda_{n_i} \end{pmatrix} \end{aligned} \quad (3.4)$$

Using this approach, a collection of such PHs is available at the start of our recursive scheme to model each activity A_i in every processor-submodel.

PH Model of Call Response Times

Some activities issue calls to entries of other service centers. Let $(\boldsymbol{\alpha}_{s,e}, \mathbf{T}_{s,e})$ represent the response time of a service center s upon invocation of its entry e and let n_{calls} be the number of requests to e . If n_{calls} is deterministic, we convolve n_{calls} PHs $(\boldsymbol{\alpha}_{s,e}, \mathbf{T}_{s,e})$. If n_{calls} is stochastic, we assume it to be geometrically distributed with parameter $p_{n_{calls}}$. We model this process by

a CTMC which has the same structure as that in Figure 3.5 and representation

$$\boldsymbol{\alpha} = (1 - p_{n_{calls}})\boldsymbol{\alpha}_{s,e}, \quad \mathbf{T} = \mathbf{T}_{s,e} + (1 - p_{n_{calls}})(-\mathbf{T}_{s,e}\mathbf{1}\boldsymbol{\alpha}_{s,e}) \quad (3.5)$$

Sequence Pattern

If an activity graph contains a linear sequence of contiguous activities A_i and A_j , with PH distributions $(\boldsymbol{\alpha}_i, \mathbf{T}_i)$ and $(\boldsymbol{\alpha}_j, \mathbf{T}_j)$, then we can merge their PH distributions into a single PH obtained by convolution. This can be done by direct application of the following result [Neu94].

If X_i and X_j are statistically independent random variables with PH distributions $(\boldsymbol{\alpha}_i, \mathbf{T}_i)$ and $(\boldsymbol{\alpha}_j, \mathbf{T}_j)$, then $X_i + X_j$ is a PH random variable $(\boldsymbol{\alpha}, \mathbf{T})$, where

$$\begin{aligned} \boldsymbol{\alpha} &= (\boldsymbol{\alpha}_i, (1 - \boldsymbol{\alpha}_i\mathbf{1})\boldsymbol{\alpha}_j), \\ \mathbf{T} &= \begin{pmatrix} \mathbf{T}_i & -\mathbf{T}_i\mathbf{1}\boldsymbol{\alpha}_j \\ \mathbf{0} & \mathbf{T}_j \end{pmatrix} \end{aligned} \quad (3.6)$$

By recursively traversing the graph is possible to iteratively apply this expression to merge all linear sequences of activities into a single PH, provided that each of them does not include any other loop or selection pattern. If any of the latter patterns are present, we first need to apply suitable modifications to the PH models to account for them.

Selection Pattern

The mixture operation is performed to do PH aggregation for a selection with k branches. In such a case, the algorithm needs k PH representations $(\boldsymbol{\alpha}_i, \mathbf{T}_i)$, $i = 1, \dots, k$, to represent the response time distribution along each branch. The following PH mixture formula is then readily applied to replace the k PH with a single aggregate PH for the branch [Neu94].

A finite mixture of PH random variables $\sum_{i=1}^k p_i X_i$, where $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$, is a PH

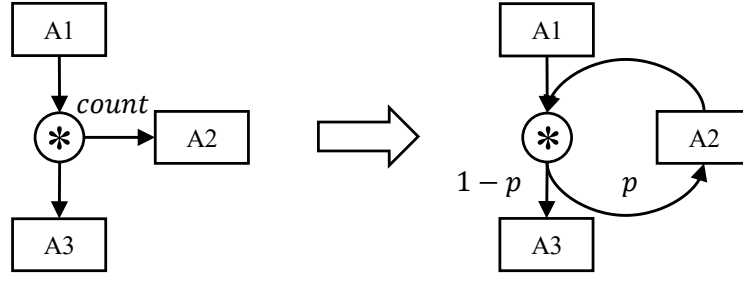


Figure 3.6: Loop transformation.

random variable $(\alpha, \mathbf{T})$, where

$$\begin{aligned}\alpha &= (p_1\alpha_1, p_2\alpha_2, \dots, p_k\alpha_k), \\ \mathbf{T} &= \text{diag}(\mathbf{T}_1, \mathbf{T}_2, \dots, \mathbf{T}_k)\end{aligned}\tag{3.7}$$

A complication arises when considering selection patterns where the activities along a branch are served in different submodels. Suppose, for example, that a selection pattern in processor-submodel s has two branches, i and j , the first leading to a call to entry e_i in some task-submodel t and the second for an activity A_j , executing at the service center of s . Then, the mixture that determines the conditional response time W_c^e is calculated using PHs for the two branches computed at the respective submodels where the branch is executed. That is, we need to retrieve different PHs for the branches and assemble them into a single PH for W_c^e using a dedicated procedure on top of the ones that compute PHs in the processor-submodel s and in the task-submodel t .

Loop Pattern

Loop patterns are handled after a pre-processing step. We approximately replace a loop that performs an average repeat count (cnt) with a probabilistic self-loop as shown in Figure 3.6. This implies that we approximate the loop with a geometric distribution with probability $p = cnt/(cnt + 1)$, which has mean cnt , and then use p as the looping probability.

After this modification, assume that the activity (or subset of the activity graph) on which we loop on is characterized by a PH distribution $(\alpha_i, \mathbf{T}_i)$. As shown in Figure 3.5, we modify the

PH representation to account for looping as follows

$$\begin{aligned} \boldsymbol{\alpha} &= p\boldsymbol{\alpha}_i, \\ \mathbf{T} &= \begin{pmatrix} -\lambda_1 & \lambda_1 & \cdots & 0 \\ 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ p\alpha_1\lambda_{n_i} & p\alpha_2\lambda_{n_i} & \cdots & -\lambda_{n_i} + p\alpha_{n_i}\lambda_{n_i} \end{pmatrix} \end{aligned} \quad (3.8)$$

where $\mathbf{T}$ can be summarized as $\mathbf{T} = \mathbf{T}_i + p(-\mathbf{T}_i\mathbf{1}\boldsymbol{\alpha}_i)$.

Nested Loops

The procedure to analyze graphs with nested loops is similar, but requires additional checks to detect and aggregate the subgraph that lies within each loop. That is, given two subgraphs G and G' describing two loops in the CTMC, in our LQNs these are either disjoint or one fully contains the other, e.g., $G \subseteq G'$. Assume that the innermost loop, in our example G , does not have other loops inside. It then consists only of sequence and branch patterns, thus it can be merged into a single PH using the rules for these patterns, after which the loop pattern around G can be eliminated using (3.8). By repeatedly applying this technique, PHs for activity graphs with loop patterns are progressively obtained by eliminating the loops one-by-one.

3.3.5 End-to-End Response Time PH Computation

Using the methods proposed in the previous subsection, we can generate the PH for the host demand at a node, and use this with MDNs to predict the response time distribution of the task client entries. Thus, as shown in Figure 3.3, PHs progressively become available to describe response time distributions at all the entries of all tasks. In this section, we describe the recursive combination of these PHs to obtain the end-to-end response time prediction.

The process begins from the bottom-most layers, which are the processor-submodels. In these layers, host demands are given and can be aggregated into PHs based on the activity graph patterns. However, some passages at the delay node may involve spending time in calls to

remote entries. Because processor-submodels are processed first, we do not yet have at this stage PH distributions that describe the response time distribution of such calls. To obviate the need for analyzing layers multiple times in an iterative fashion, we assume at these submodels that such calls follow exponential distributions with means matching the AMVA solution of the LQN, which is an input to our methodology. This assumption tends to have a marginal effect on the response time distribution of the submodel, partly because these response time components are accumulated at the delay node.

After applying Algorithm 4, the same process is then applied in task-submodels immediately above the processor-submodels. Also in this case, when a response time distribution for a call is not yet available, being associated to a layer yet to be processed, it is assumed to be exponentially distributed with mean obtained by AMVA. The process for computing a PH for the host demand is then iterated, together with the MDN prediction and the generation of PHs for the conditional response times of the entries of client tasks of the task-submodel. Thus, by recursively applying this analysis method, we ultimately reach the uppermost layers, i.e., the reference tasks, obtaining the desired distribution of the end-to-end response time W_r for every entry r at the reference task, which completes the analysis.

3.4 Evaluation

3.4.1 MDN Submodel Analyzers

In this section, we first evaluate if MDNs are capable of accurately predicting unknown response time distributions. We refer to a single-class or two-class input vector $\mathbf{x}$ as a pair. We generate training and test pairs according to Table 3.2 and employ the discrete-event simulator in JMT 1.2.1 [BCS09], instantiated via the LINE tool [Cas20], to obtain response times under FCFS or PS discipline. A simulation submodel is defined as a pair with a scheduling discipline. For each job class in a simulation submodel, we tag one of its jobs; then we record its arrival and departure time at the service center to measure its response time at each passage. We take all

Table 3.2: Submodel parameters.

Symbol	Parameter	Range
N	Population	1-100
Z	Think time	0.1-10
D	Host demand mean	0.05-2
SCV	Host demand squared coefficient of variation	0.1-2.5
m	Number of servers	1-32

response times of this tagged job. These response time samples collectively form the empirical distribution of the response time of its job class. For each submodel type, we train a separate MDN analyzer using 20000 pairs and their response time distributions.

We consider two performance metrics: *mean absolute percentage error* (MAPE) and *mean Kolmogorov-Smirnov distance* (MKS). The first metric is defined as

$$\text{MAPE} = \frac{100\%}{N_{test}} \sum_{n=1}^{N_{test}} \left| \frac{P_n - G_n}{G_n} \right| \quad (3.9)$$

where P_n and G_n are the prediction and ground-truth values, and N_{test} is the number of test pairs. We calculate MAPE for mean, standard deviation, and 95th percentile of response times, respectively.

The Kolmogorov-Smirnov distance [MJ51] is defined as the maximum vertical distance between two cumulative density functions (CDFs), thus our second metric is

$$\text{MKS} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} \max_x |F_n^G(x) - F_n^P(x)| \quad (3.10)$$

where $F^G(x)$ is the empirical CDF from observed samples and $F^P(x)$ is the CDF of a mixture model from prediction.

We use the neural network structure depicted in Table 3.3, and only tune the number of components and kernel type. Therefore, for each of the four considered types of analyzers, we train neural networks with six settings and select the one with the lowest validation loss. We train each network for 50 epochs using a batch size of 5,000 observed samples, with a validation

Table 3.3: Neural network architecture and optimizer.

Hyperparameter	Values
Number of hidden layers	3
Hidden layer neurons	64
Activation function	ReLU
Number of components I	3, 5, 7
Kernel type	Gaussian, Gamma
Optimizer	RMSprop
Learning rate α	0.001
Decay factor of α	50% per 10 epochs

Table 3.4: Minimum validation loss of MDNs.

Kernel	Gaussian			Gamma		
Scenario/ I	3	5	7	3	5	7
FCFS, 1 class	1.616	1.590	1.583	1.578	1.582	1.577
FCFS, 2 classes	2.318	2.315	2.314	2.336	2.326	2.322
PS, 1 class	2.184	2.127	2.110	2.096	2.097	2.095
PS, 2 classes	3.196	3.151	3.139	3.129	3.127	3.127

split of 0.05. As shown in Table 3.4, all selected analyzers have seven components. We refer to the four optimal MDNs in top-down order as MDNs I, II, III, and IV. The analyzers I, II, and IV are with Gamma kernels, while the analyzer III adopts Gaussian kernels. We can observe that, for PS scheduling models, an MDN with Gamma kernels has a lower loss compared to that with the same number of Gaussian kernels. Then, our analyzers are evaluated on test sets (5000 pairs); the results are shown in Table 3.5. As can be observed, all MAPE values are below 6%, i.e., the trained MDNs show an overall good performance. Besides the evaluations above, we can visualize the prediction performance via different types of plots. Figure 3.7a shows that the vast majority of KS distances are below 0.10, which means MDNs get very close to the ground-truth response time distributions. Meanwhile, we can observe that PS cases are simpler for neural networks to approximate with regard to the KS distance. Figure 3.7b gives a comparison between simulations and sampling distributions from MDN predictions. In this experiment, we simulate a pair and obtain 1758 and 1781 response time samples from FCFS and PS queues, respectively, then we employ MDN I and II to make predictions and generate the same number of samples from estimated distributions. As can be seen, the trained MDNs cope well with representing unknown distributions. The next experiment is to compare mean,

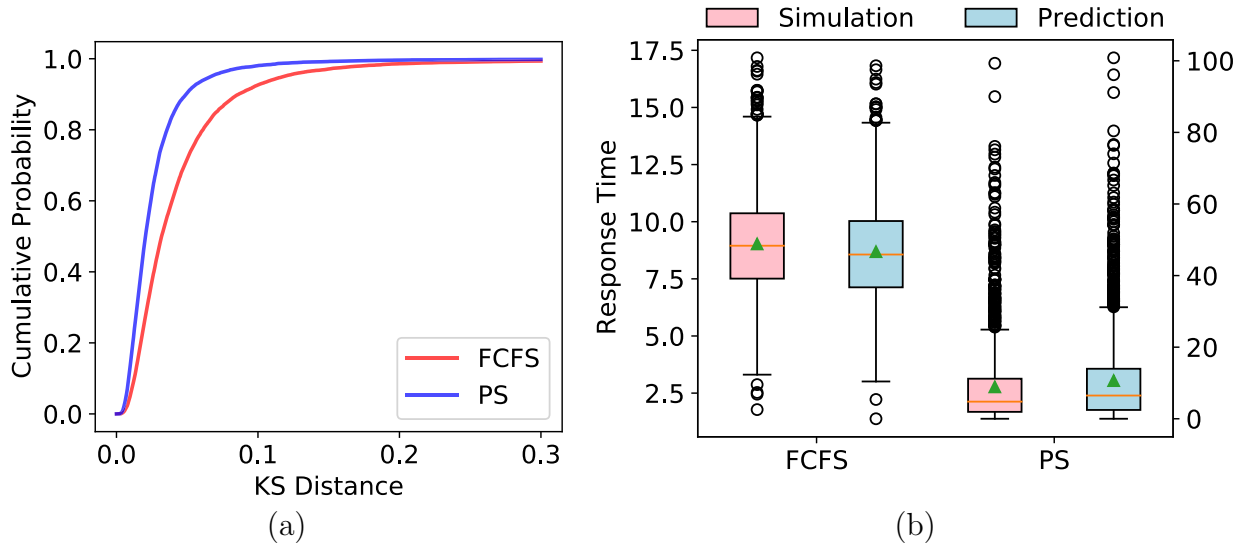


Figure 3.7: (a): The CDFs of KS distances between simulations and predictions for all FCFS and PS test pairs. (b): Summary statistics (the orange line and green triangle inside box indicate the median and mean value) on the response time given $\mathbf{x} = (\{85, 8, 0.4, 1.5\}, 2)$. The left and right y-axis represent the response time under FCFS and PS, respectively.

Table 3.5: The MAPE and MKS results of MDNs.

Analyzer	Scenario	MAPE			MKS
		Mean	Std	95th	
MDN I	FCFS, 1 class	3.0%	4.4%	3.2%	0.036
MDN II	PS, 1 class	4.6%	5.6%	5.2%	0.023
MDN III	FCFS, 2 classes	3.6%	5.5%	3.5%	0.055
MDN IV	PS, 2 classes	5.0%	5.8%	5.5%	0.031

variance, and percentiles between simulations and predictions by varying the values of distinct input variables. Figure 3.8a to Figure 3.9b shows sensitivity analyses where we can observe that our MDNs give acceptable approximations over varying values of population, number of servers, think time, and host demand mean. This means MDNs are capable of discriminating between the different variables and calculating how each type influences the response time behavior. Finally, we visualize the prediction performance in a multiclass submodel with FCFS scheduling discipline and different service rates. This is the situation where product-form assumptions do not hold for the underlying queueing network. Figure 3.9c to 3.9d display two predictions that are below the average level. Their KS distances are 0.07 and 0.12, around 1.3 and 2.2 times longer than the MKS value, 0.055, but the overall trends of predictions are still reasonable.

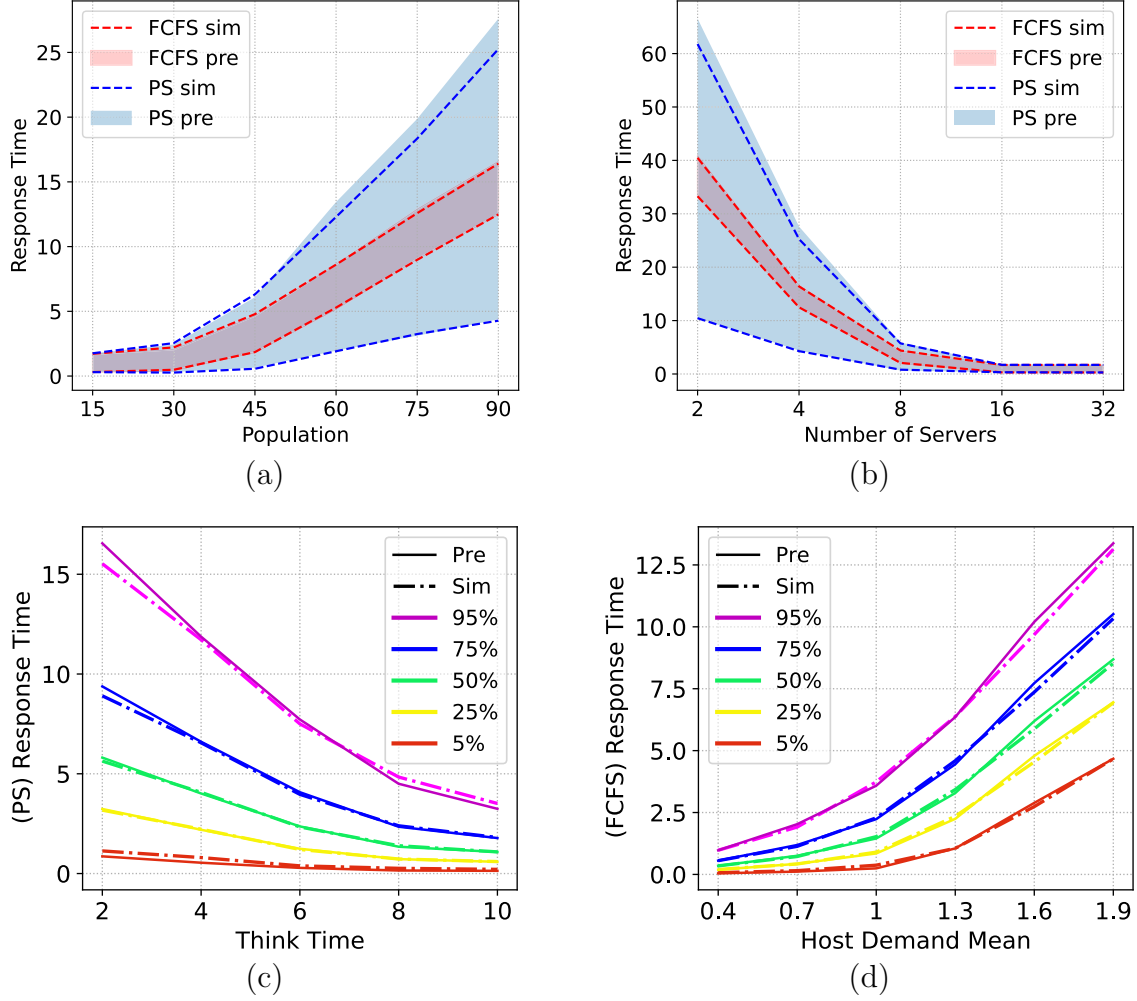


Figure 3.8: (a)-(b): Mean $\pm$ one standard deviation (shaded area) of the response time. Starting with $\mathbf{x} = (\{90, 8, 1, 0.5\}, 4)$, varying population and number of servers. (c)-(d): Percentiles of the response time. Starting with $\mathbf{x} = (\{35, 8, 1, 0.5\}, 4)$, varying think time and host demand mean. Sim and Pre represent simulation and prediction.

3.4.2 LQN Response Time Prediction

We conduct evaluations on the approximation of end-to-end response time distributions on two LQN topologies: E-commerce [CDMI11] and an interconnected network. For each case, 500 model instances are randomly generated using the parameters in Table 3.6. The end-to-end response times are estimated by our framework and a baseline, which we choose to be LQNS 6.2 [FAOW⁺08], [FMW⁺05], the de facto standard analytical tool for LQN models. Deterministic calls are specified via the *call order* option in the LQNS solver. The accuracy of an estimation is measured by means of the absolute percentage error (APE) with respect to the simulation result obtained by LQSIM tool [FMW⁺05], i.e.

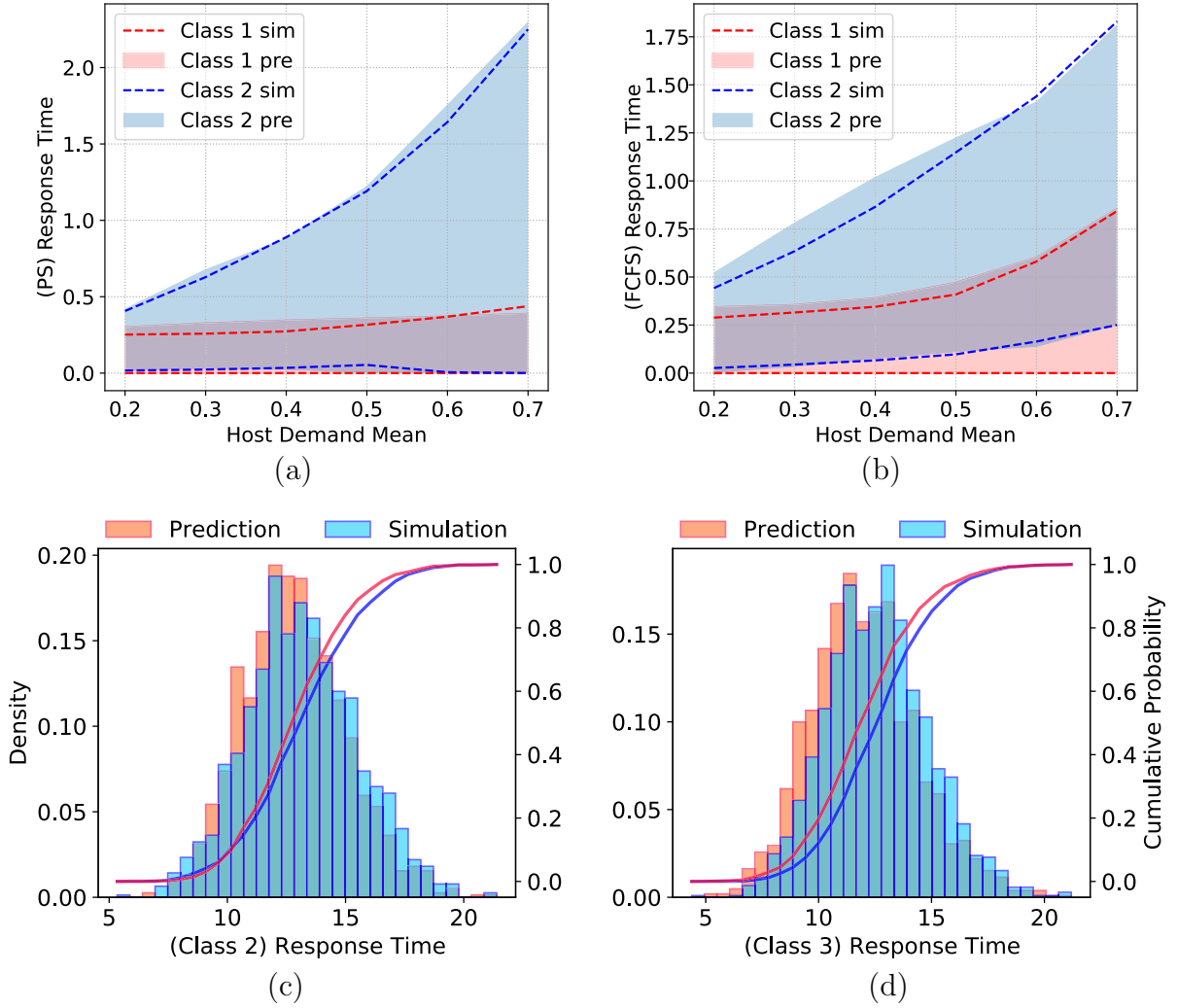


Figure 3.9: (a)-(b): Mean $\pm$ one standard deviation (shaded area) of the response time. Starting with $\mathbf{x} = (\{25, 3, 0.1, 2\}, \{30, 7, 0.2, 0.8\}, 4)$, varying host demand mean of class 2. (c)-(d): Analyzing a submodel $\mathbf{x} = (\{15, 3, 0.7, 1\}, \{25, 10, 0.8, 0.6\}, \{30, 9, 0.1, 1\}, \{20, 5, 0.3, 1.4\}, 2)$ under FCFS.

$$\text{APE} = \left| \frac{P - G}{G} \right| \times 100\% \quad (3.11)$$

where P and G represent the prediction and ground-truth value. Because we estimate response times for a batch of model instances, we obtain a set of absolute percentage errors, which collectively give an absolute percentage error distribution. The APE distribution can be characterized using percentiles, which we show in our results. Both our framework and LQNS can provide accurate estimates of mean response times, their absolute percentage errors are less than 10%. In terms of density estimates, the proposed framework shows a visible improvement over the baseline.

Figure 3.10 displays the topology of the E-commerce system. Users are classified into three categories according to the behavior in *CustomerProcess* task. The activity graphs (a), (b), and (c) shown in Figure 3.11 are assigned to the entry *browse*, *cart*, and *placeOrder* respectively, service requests are sent by the dashed activities (A3→catInfo, A4→update, A6→delivOrder, A9→custInfo, A13→cartInfo, A15→empty). Table 3.7 reports MAPE and APE distribution of standard deviation (σ) of the response time across 500 model instances. The i th percentile is the value such that $i\%$ of the APEs are lower than this value. We discuss results by dividing all model instances into four groups with different ranges of system population. As can be observed, the reduction of average error by our framework is between 85% and 95% compared to the baseline. Our median absolute percentage errors are at most 0.07 while the median errors given by LQNS reach around 0.90. We interpret this to be due to the general difficulty of estimating response time distributions using analytical methods, as remarked in Section 2.4. The percentile statistics show that error distributions of our framework are positively skewed, but those of LQNS are negatively skewed. The numerical results indicate that the proposed framework is effective and significantly outperforms LQNS in estimating response time densities in models incorporating activity graphs.

Next, we apply our framework to analyzing the interconnected network shown in Figure 3.12. To help the visualization, the figure does not show host processors. There are two shared processors, one is for R1 and R2, and the other is for R3 and R4. The remaining tasks have their own host processors. The main feature of this network is multichain interconnection, which requires estimating response time densities in a set of multiclass queues. Figure 3.13 compares the prediction performance of our framework and the state-of-the-art. As observed in Figure 3.13a and Figure 3.13b, our framework can accurately estimate densities in multiclass LQN models, the error reduction is over 75% compared to LQNS. Then, we visualize the prediction performance on distinct population groups in Figure 3.13c and Figure 3.13d. As the system population gets larger, the absolute percentage errors given by the baseline show a quick increase. In contrast, we can observe that with our framework the errors remain on a stable level.

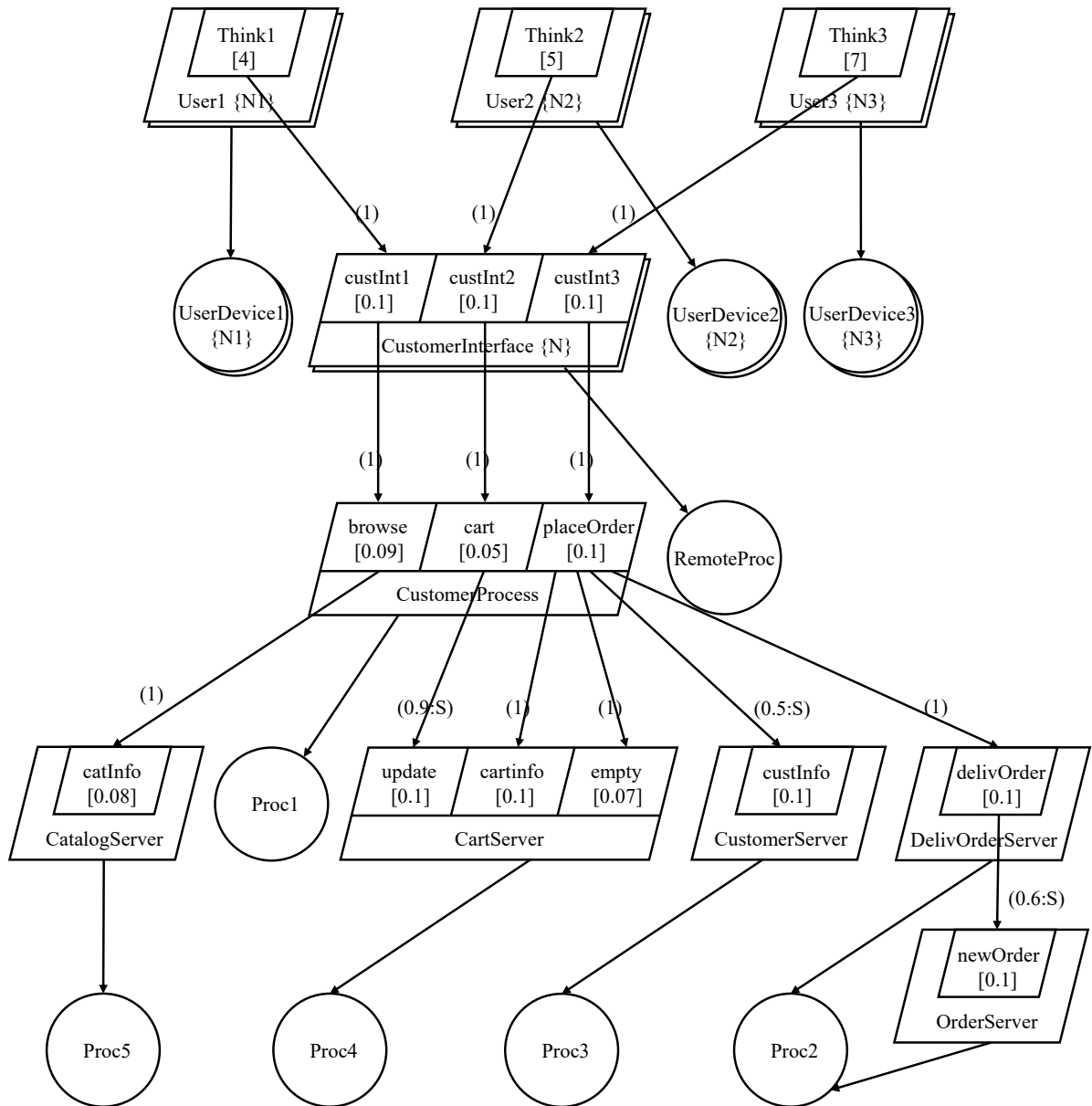


Figure 3.10: LQN model of an E-commerce system.

Table 3.6: Case study parameters.

Parameter	Range
System population	20-100
Think time	3-10
Multiplicity	1-8, ∞
Activity host demand mean	0.05-0.1
Activity host demand squared coefficient of variation	0.4-1.2
Number of deterministic calls	1, 2
Number of stochastic calls	0.1-2
Loop counts	0.1-2.5

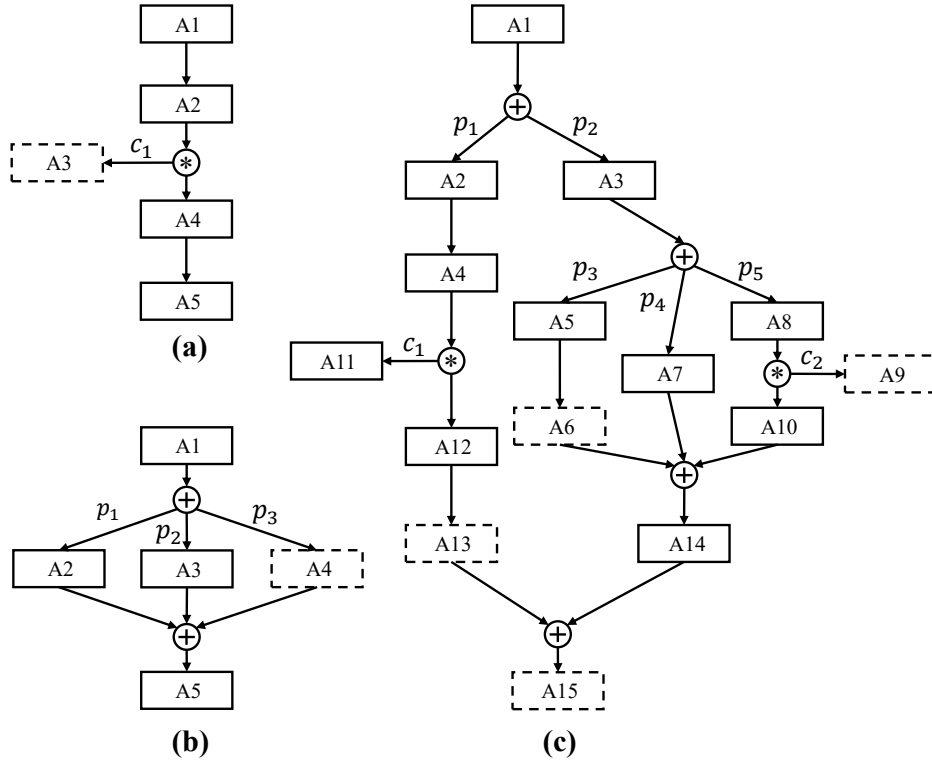


Figure 3.11: Three activity graphs used in the case study of Figure 3.10.

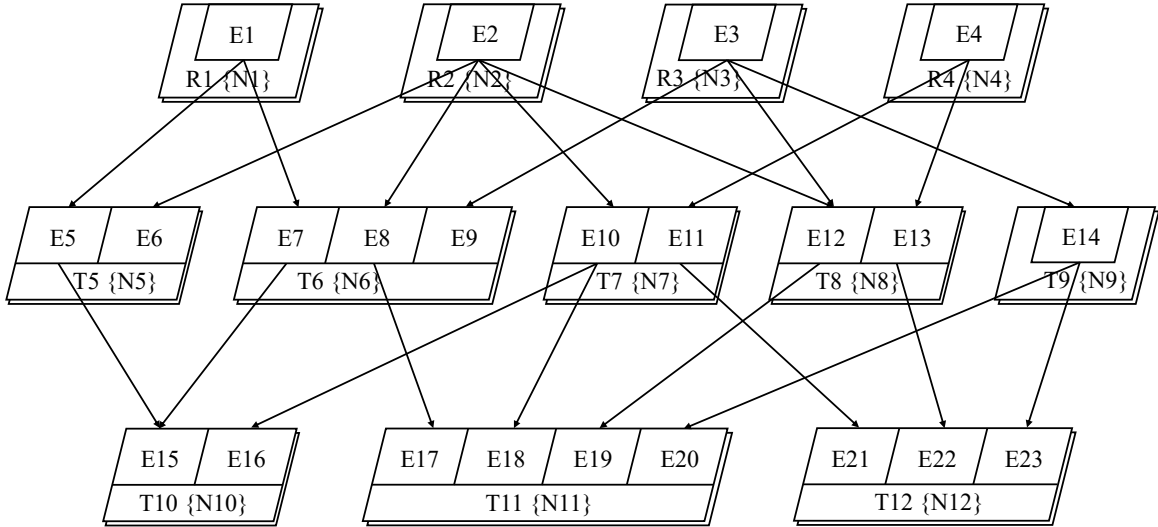


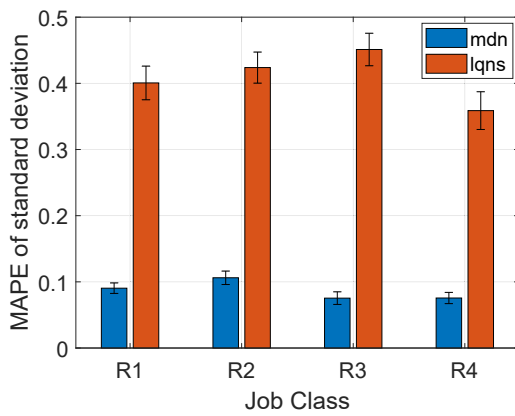
Figure 3.12: LQN model of an interconnected system.

3.4.3 Layering Depth Experiment

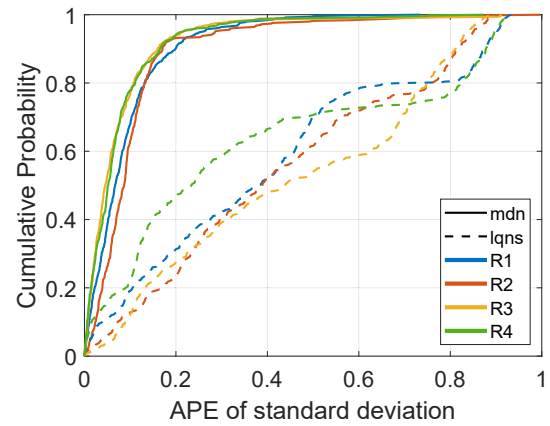
Getting accurate estimations with deeper systems may pose challenges due to the accumulation of errors during the propagation of density across layers. To assess the performance of the proposed scheme, we generate 300 deeper LQN models than the ones considered before, with varying depth (5-10 layers) and using the parameters listed in Table 3.8. The remaining

Table 3.7: Percentiles of APE distributions and MAPE of standard deviation across 500 instances with topology as in Figure 3.10.

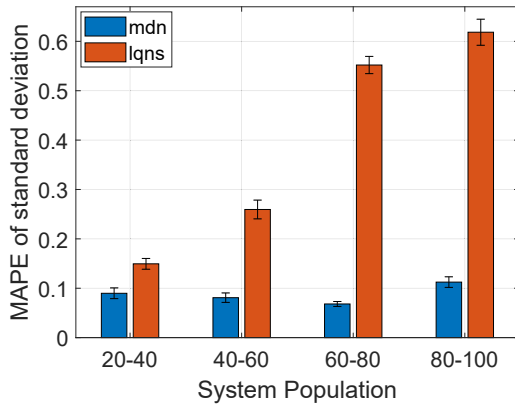
System Population		5th	Med	95th	99th	Avg
20-40 (112 instances)	lqns	0.143	0.881	0.933	0.945	0.763
	mdn	0.006	0.070	0.308	0.680	0.105
40-60 (140 instances)	lqns	0.631	0.905	0.944	0.956	0.878
	mdn	0.004	0.041	0.207	0.541	0.066
60-80 (121 instances)	lqns	0.872	0.915	0.952	0.978	0.892
	mdn	0.005	0.049	0.254	0.567	0.080
80-100 (127 instances)	lqns	0.840	0.923	0.962	0.983	0.897
	mdn	0.004	0.060	0.417	0.639	0.104



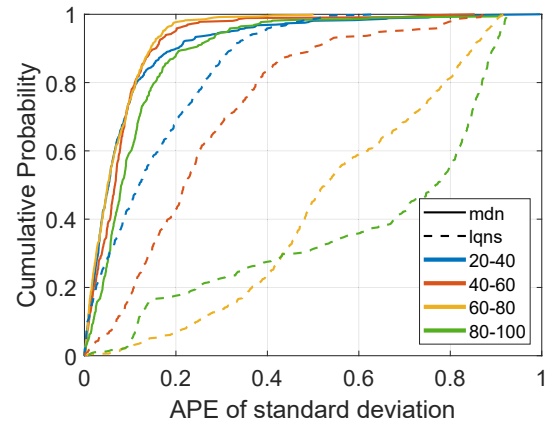
(a)



(b)



(c)



(d)

Figure 3.13: MAPE and APE distribution across 500 instances with topology as in Figure 3.12. Here, error bars represent the 95% confidence interval. It can be observed that the confidence intervals are relatively narrow.

parameters are consistent with those in Table 3.6. Calls in the models do not jump over layers. We re-train the MDNs with datasets incorporating a wider range of squared coefficient of variation (0.001-20). Before use, users need to train their models on a dataset whose parameters

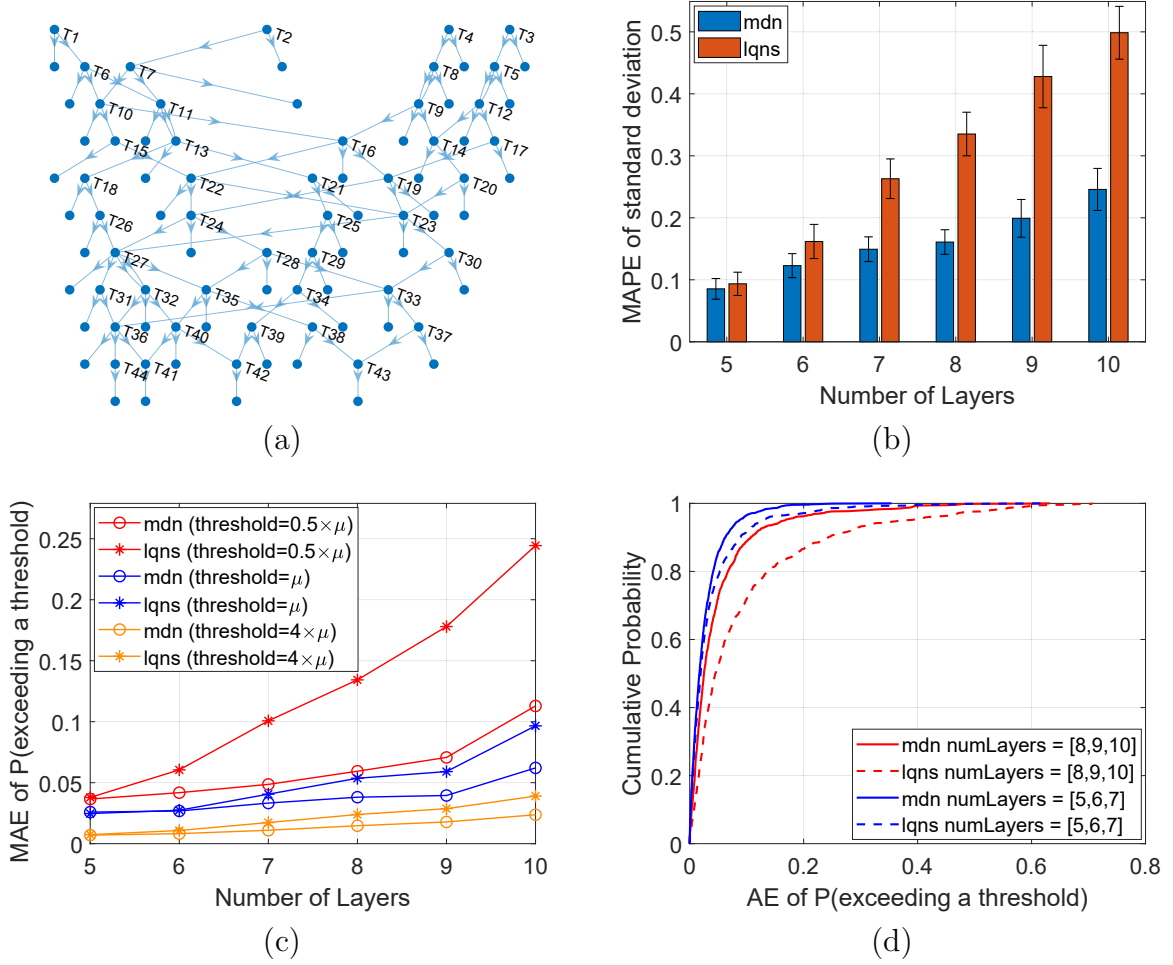


Figure 3.14: (a): Example of a randomly generated LQN model, for ease of presentation, entry and activity details are omitted. (b)-(c): MAPE and MAE over different number of layers. The bar diagram shows 95% confidence intervals. (d): AE of the exceedance probability categorized into three groups based on number of layers.

Table 3.8: Model size parameters.

Parameter	Value
Maximum number of layers	10
Maximum number of tasks per layer	5
Maximum number of entries per task	2
Activity patterns of entries	sequence, loop, selection
Maximum host demand mean per activity	0.0015
Maximum host demand squared coefficient of variation per activity	10

are valid for the target problems. We evaluate the estimated response time distribution by comparing the probability of exceeding a threshold with the simulation result. The exceedance probability can be obtained via the *max-service-time* option in the LQNS solver. For each

model, three thresholds are set based on mean response times returned by LQNS: half of the mean response time, the mean response time, and four times the mean response time. Two metrics are employed to evaluate the estimated exceedance probability: absolute error (AE) and mean absolute error (MAE), which are computed analogously to APE and MAPE but substitute $|P - G|/G$ with $|P - G|$, where P is the estimated exceedance probability and G is the probability determined by LQSIM. The results are shown in Figure 3.14, which indicate lower errors for the proposed method than LQNS, with the gap growing as more layers are added. The proposed method is also more precise in estimating the probability of exceeding a response time threshold. In addition, the execution of density propagation and MDN predictions is highly time-efficient, with every tested model completed in less than 10 milliseconds on a commodity desktop.

3.5 Summary

In this chapter, we propose a novel approach that combines a PH analytical framework with MDNs to tackle the challenge of characterizing response time distributions in LQNs abstracted from distributed systems with varying topologies. We summarize the main points as follows:

- Our approach applies SRVN layering strategy to decompose an LQN into a set of machine repairman queueing networks where we characterize service demands using PH distributions and predict response time distributions by MDNs.
- We propose a way to characterize the time spent in a loop using PH random variables, and apply PH distributions to represent service demands of arbitrary activity graphs that consist of sequences, selections and loops.
- We present a PH modeling approach for the durations of service requests to propagate density estimations across layers, which forms an end-to-end PH analytical framework that enables the tractability of response time distributions.

- We show that MDNs yield highly accurate predictions of response time distributions in machine repairman queueing networks. Additionally, we find that Gamma kernels enable an MDN to better fit response times under PS scheduling.
- We embed trained MDNs within our PH analytical framework and evaluate the prediction performance using LQNs with complicated activity graphs and interdependencies. Our approach significantly reduces the density estimation error by over 70% compared to the state-of-the-art analytical method.
- We conduct depth experiments and compute the probability that the response time exceeds a specified threshold. The results show that our approach accurately estimates exceedance probabilities and becomes more reliable to use than analytical method as the number of layers increases.

Chapter 4

Graph Representation Learning for Evaluating Closed Queueing Networks

4.1 Introduction

Queueing networks serve as a popular performance model in the analysis of business processes and distributed computing systems. Solving queueing network models and obtaining performance estimates support the decision-making of system designers. Conventionally, the research on evaluating queueing networks primarily focuses on deriving closed-form analytical solutions. However, many problems in queueing theory remain unresolved, with their analytical and numerical solutions proving to be challenging using conventional methods, such as those failing to meet the assumptions of the BCMP theorem [BGDMT06]. Mean Value Analysis (MVA) and its extensions are perhaps the most widely used algorithms in queueing theory. However, they are primarily designed for solving closed product-form queueing networks. Conventional analytical methods can lead to inaccurate or misleading results when applied to networks with non-product-form characteristics (e.g., class-dependent service times at FCFS service center, blocking networks), which are often encountered in modeling real systems.

The advancement of machine learning has drawn growing attention from the queueing community, as this technique is not constrained by the strict assumptions of conventional methods that

make problems manageable. In principle, the limitations of conventional analytical methods can be overcome through the proper use of machine learning approaches. This may enable us to tackle complex queueing networks that conventional methods struggle with. Learning-based approaches are therefore increasingly applied into queueing research to generalize the performance evaluation capability. Existing research can be categorized into three main directions. The first uses machine learning methods to predict the performance of complex queues. The second simulates real queueing systems using generative neural networks to overcome the barriers of expensive simulations. The third employs reinforcement learning approaches for control and optimization in queues. However, these explorations tend to focus on fixed systems with specific network topologies. There is a scarcity of work on developing a learning-based model to serve as a general-purpose solver for various queueing networks.

In this chapter, we propose a GNN model designed to address queueing problems, which enables the solution of a broader range of closed queueing networks involving non-product-form characteristics. The experimental results demonstrate that our model can effectively evaluate the performance of closed queueing networks with flexible topologies and varying numbers of job classes. This research reveals the potential of learning-based models as a general-purpose solver for different queueing networks.

The rest of the chapter is organized as follows: Section 4.2 formulates a class of general closed queueing networks we will solve. Section 4.3 proposes our GNN model and its detailed message-passing mechanism. Section 4.4 evaluates the prediction performance of our model and demonstrates its advantages. Section 4.5 draws the conclusions of this chapter.

4.2 Problem Formulation

Because the GNN approach can accommodate inputs with flexible shapes, this makes it possible for us to solve more general closed queueing networks without restrictions on topology or the number of job classes. Here, we formulate a class of closed queueing networks for our analysis. Suppose a closed queueing network comprising one delay station and multiple FCFS service

centers indexed by $i = 1, \dots, K$, which accommodates job classes labeled as $r = 1, \dots, C$. The delay station is used as the reference station for all job classes. The workload of job class r is defined by its population, denoted by N_r , and its think time at the delay station, characterized by the mean Z_r and the squared coefficient of variation $Zscv_r$.

A class- r job is served sequentially by K_r out of the total K service centers (i.e., $1 \leq K_r \leq K$), with each service center denoted by $S_{i,r}$. In a closed queueing network, multiple job classes can share the same service center. As a result, a service center i may have different $S_{i,r}$ notations (e.g., both $S_{2,1}$ and $S_{2,4}$ refer to the second service center). The service center $S_{i,r}$ possesses m_i servers, and the service times of class- r jobs at $S_{i,r}$ are characterized by the mean $D_{i,r}$ and $Dscv_{i,r}$. We aim to develop a GNN model that predicts local performance measures per class at each service center $S_{i,r}$ for a given closed queueing network. The local performance measures of interest include throughput $X_{i,r}$, queue length $Q_{i,r}$, utilization $U_{i,r}$, and response time $R_{i,r}$. In the long run, when a network enters the steady state, the throughput of a job class is uniform across all associated service centers, i.e., $X_r \equiv X_{1,r} \equiv \dots \equiv X_{K,r}$. For this reason, we refer to throughput as the global performance measure.

4.3 Methodology

4.3.1 Performance Evaluation using MLPs

In this chapter, we aim to develop a learning-based model that can serve as a general-purpose solver for the performance evaluation of various closed queueing networks. We identify the GNN approach as a possibility for achieving this. Before delving into it, we will temporarily focus on another method, MLPs, known for their strong fitting capabilities in regression tasks. The universal approximation theorem [HSW89] provides a theoretical assurance that an MLP is capable of showing flexible expressiveness. In theory, a three-layer neural network can approximate any functions [Ism23]. In our context, the input to an MLP is a set of parameters that characterizes the workload and resource demand in a queueing network, and the output

is the performance measure of interest. However, the input shape of an MLP is fixed, which limits its ability to deal with a wide range of closed queueing networks with flexible routings and varying numbers of job classes. Therefore, we apply the MLP method to solve a fixed but a classic queueing model known as the machine repairman network [BGDMT06], which is a closed queueing network comprising a delay station and a service center. It also acts as the basic submodel decomposed from layered queueing networks [WNPM95].

Due to the constraint of a fixed input shape, we use MLPs to assess only two-class machine repairman queueing networks with an FCFS scheduling discipline and class-dependent service times. This represents a non-product-form case that BCMP theorem cannot address [BCMP75]. The MLP is a function of machine repairman model parameters comprising population, think time and service time per job class, and the number of servers at the service center. Because the variability in job sizes has an important impact on performance measures, we consider both the mean and squared coefficient of variation of the think time and service time. We aim to obtain the throughput X_r , utilization U_r , mean response time R_r , and mean queue length Q_r per job class at the service center. We set the output of the MLP as mean response time R_r , and compute the remaining measures from it.

Given the prediction for the mean response time of class- r jobs at the service center, the throughput can be obtained by applying Little's law

$$X_r = \frac{N_r}{R_r + Z_r}, \quad (4.1)$$

where N_r and Z_r denote the population of class- i jobs and their mean think time, respectively.

Then the utilization can be computed by

$$U_r = \frac{X_r D_r}{m}, \quad (4.2)$$

where D_r and m denote the mean service time of class- r jobs and the number of servers at the service center, respectively.

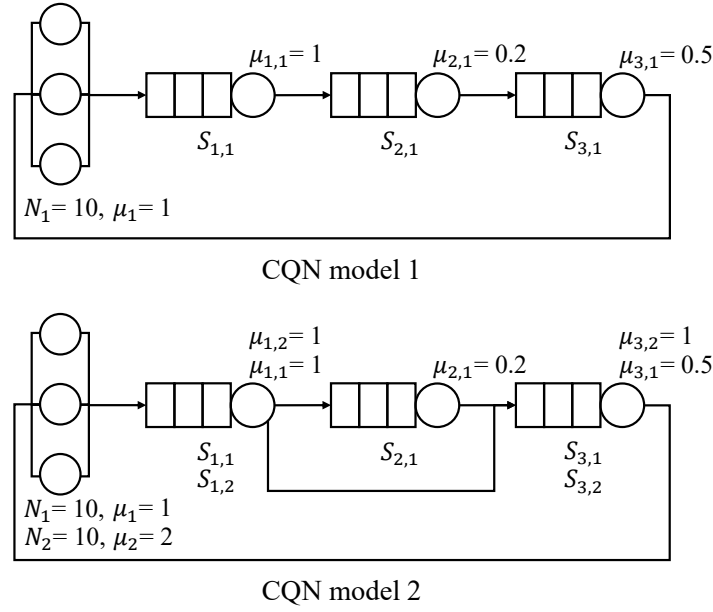


Figure 4.1: Motivating examples underlying the rationale for our GNN design.

Applying Little's law one more time, the mean queue length at the service center is

$$Q_r = X_r R_r = \frac{N_r R_r}{R_r + Z_r}. \quad (4.3)$$

4.3.2 Underlying Motivations for the GNN Design

In this section, we discuss the two key motivations shaping the architecture of our GNN model.

(1) Each queueing network has at least one bottleneck limiting the overall performance. In other words, while the execution of a job class may be associated with several service centers, its performance primarily depends on the bottleneck service center. For example, in a closed single-class queueing network, the service center with the minimum service rate will be the limiting factor in the throughput. Therefore, in our analysis for a job class, we assign weights to its service centers to characterize the impacts of potential bottlenecks.

Motivating example 1: This first example presented in Figure 4.1 is a closed single-class tandem queueing network with one reference delay station and three service centers $S_{1,1}$, $S_{2,1}$, and $S_{3,1}$. There are 10 jobs circulating within the network, and the scheduling discipline is FCFS. The think time and service time distributions are exponential, with a rate of 1 for think

Table 4.1: Performance measures of class-1 jobs in motivating examples shown in Figure 4.1.

	CQN model 1				CQN model 2			
	$Qlen$	$Util$	$Respt$	$Tput$	$Qlen$	$Util$	$Respt$	$Tput$
$S_{1,1}$	0.25	0.20	1.25	0.20	0.82	≈ 0.20	4.10	≈ 0.20
$S_{2,1}$	8.89	1	44.45	0.20	6.09	≈ 1	30.45	≈ 0.20
$S_{3,1}$	0.66	0.40	3.30	0.20	2.89	≈ 0.40	14.45	≈ 0.20

time, and rates of 1, 0.2, and 0.5 for service times at $S_{1,1}$, $S_{2,1}$, and $S_{3,1}$, respectively. The service center $S_{2,1}$, with a rate of 0.2, is thus the bottleneck. The performance measures of this example are presented in Table 4.1. It can be observed that the longest queue forms at $S_{2,1}$, and as expected, the system throughput is 0.2. This center plays a dominant role in determining the system behavior and should therefore be weighted accordingly.

(2) The performance of a job class at a service center depends on the state of this job class. If multiple job classes share that service center, then the performance of interest is also affected by the states of other job classes.

Motivating example 2: The second model in Figure 4.1 is a closed two-class queueing network that is built by adding 10 class-2 jobs into the first model. After departing from the delay station, the class-2 jobs receive services at the first and third service centers. Their think times and service times follow exponential distributions as well, with a rate of 2 for think time, and a rate of 1 for service times at both service centers $S_{1,2}$ and $S_{2,2}$. As can be observed from Table 4.1, the queue length and average response time of class-1 jobs at $S_{3,1}$ increases significantly from 0.66 to 2.89 and from 3.30 to 14.45, respectively, which indicates that the addition of a new job class affects the local performance of the original job class.

Meanwhile, an interesting observation is that the queue length and average response time of class-1 jobs at $S_{2,1}$ decrease. They are reduced from 8.89 to 6.09 and from 44.45 to 30.45, respectively. This is because the addition of class-2 jobs implicitly affects the performance of class-1 jobs at $S_{2,1}$. In other words, the addition explicitly changes the states of $S_{1,1}$ and $S_{3,1}$, which leads to an update of their effects on the class-1 global state. The new class-1 global state modifies the local state at $S_{2,1}$.

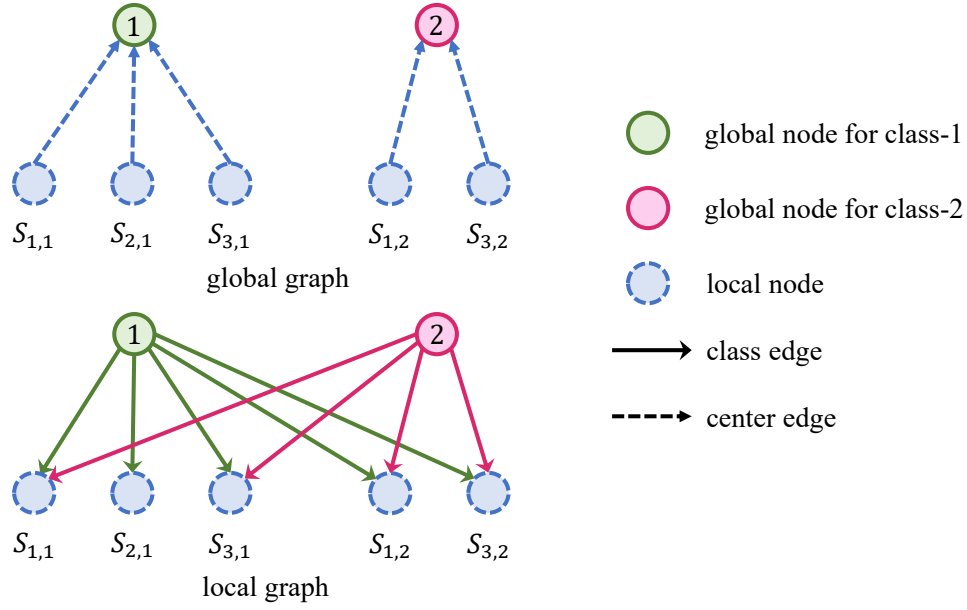


Figure 4.2: The graph representation for the motivating two-class queueing network in Figure 4.1. There are four types of elements in a graph representation: (i) global nodes: filled nodes with solid borders; (ii) local nodes: filled nodes with dashed borders (iii) class edges: solid line arrows; (iv) center edges: dashed line arrows.

4.3.3 Model Input: Graph Representation and Features

To evaluate the performance of a queueing network, we first model it as a heterogeneous graph, with nodes and edges symbolizing components and dependencies within the queueing network. The graph representation serves as the input to our GNN model. We create two types of nodes to represent the job class and service center. They are named *global node* and *local node*, respectively. Suppose a class- r job receives services at K_r service centers, then the global node for the r -th job class appears together with K_r local nodes that represent its service centers. Therefore, the graph for a closed queueing network with C job classes has C global nodes and $\sum_{r=1}^C K_r$ local nodes. Node embeddings are initialized with their input features. The features of a global node for the r -th job class comprise think time parameters Z_r , Z_{scv_r} , and the population N_r , while the features of a local node for service center $S_{i,r}$ comprise service time parameters $D_{i,r}$, $D_{scv_{i,r}}$, and the number of servers m_i . We divide the graph representation into two subgraphs, named *global graph* and *local graph*, respectively. The global graph is constructed by connecting directed edges from every local node to its matching global node, termed the *center edges*. Subsequently, the local graph is constructed by connecting directed

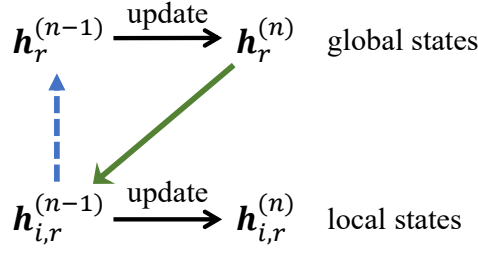


Figure 4.3: Schematic view of the state update flow in our GNN approach.

edges from each global node to all local nodes that represent the service centers where this job class receives services, termed *class edges*. An example of building a global graph and a local graph is shown in Figure 4.2.

4.3.4 The Flow of Updating Node States

Message passing is the key mechanism of GNN approaches. The objective of message passing is to encode important dependencies within the graph into node embeddings, also referred to as node states. An iteration of message passing results in an update to the state of each node. Before presenting the detailed message-passing algorithm, we provide a high-level description of the update flow for node states. At the n -th iteration of the message passing, our model first passes local node embeddings $\mathbf{h}_{i,r}^{(n-1)}$ along center edges to update global node embeddings from $\mathbf{h}_r^{(n-1)}$ to $\mathbf{h}_r^{(n)}$, then passes the newly updated embeddings $\mathbf{h}_r^{(n)}$ along class edges to update local node embeddings from $\mathbf{h}_{i,r}^{(n-1)}$ to $\mathbf{h}_{i,r}^{(n)}$. This flow is demonstrated in Figure 4.3. After iterations of state updates, the final state of each node can be used for downstream tasks such as performance prediction.

4.3.5 Message Passing on the Global Graph

The state of a job class is captured by the corresponding global node embedding. Because the throughput is a global-level performance measure, our model predicts the throughput for each job class based on its global node embedding. The system throughput is limited by the bottleneck service centers, we need to consider the impacts of distinct service center on the job

class. Therefore, we pass the local messages $\mathbf{h}_{i,r}^{(n-1)}$, $i = 1, \dots, K$ along center edges, from local nodes to a global node, and employ an attention mechanism to consider the distinct impacts of service centers. We first compute the attention score [BAY21] that characterizes the significance of a local message $\mathbf{h}_{i,r}^{(n-1)}$ to the global state $\mathbf{h}_r^{(n-1)}$

$$e(\mathbf{h}_r^{(n-1)}, \mathbf{h}_{i,r}^{(n-1)}) = \boldsymbol{\alpha}^T \sigma \left(\mathbf{W} \left[\mathbf{h}_r^{(n-1)} \parallel \mathbf{h}_{i,r}^{(n-1)} \right] \right), \quad (4.4)$$

where $\parallel$ denotes vector concatenation, $\mathbf{W}$ is a matrix of learnable weights, σ is a LeakyReLU activation function, and $\boldsymbol{\alpha}^T$ is the transpose of a vector of learnable weights.

We then normalize attention scores of local messages using the Softmax function

$$\alpha_{i,r} = \frac{\exp(e(\mathbf{h}_r^{(n-1)}, \mathbf{h}_{i,r}^{(n-1)}))}{\sum_{m=1}^K \exp(e(\mathbf{h}_r^{(n-1)}, \mathbf{h}_{m,r}^{(n-1)}))}. \quad (4.5)$$

The aggregated local message is finally obtained by performing a weighted sum

$$\mathbf{m}_{L_r}^{(n)} = \sum_{i=1}^K \alpha_{i,r} \mathbf{W} \mathbf{h}_{i,r}^{(n-1)}. \quad (4.6)$$

The global node embedding is then updated based on the aggregated local message

$$\mathbf{h}_r^{(n)} = \phi_G \left(\mathbf{h}_r^{(n-1)}, \mathbf{m}_{L_r}^{(n)} \right). \quad (4.7)$$

4.3.6 Message Passing on the Local Graph

The performance of a job class at a service center is affected by the state of the service center (i.e., busy or idle), which depends on the states of job classes that share this service center. Let $C(i, r)$ denote the set of job classes that share the service center $S_{i,r}$. In the local graph, we pass messages along class edges from global nodes corresponding to classes $s \in C(i, r)$ to the local node of $S_{i,r}$. An attention mechanism is employed to take the distinct impacts of the job classes into account.

We compute the attention score that characterizes the significance of the global messages of

$s \in C(i, r)$ to $S_{i,r}$

$$e(\mathbf{h}_s^{(n)}, \mathbf{h}_{i,r}^{(n-1)}) = \boldsymbol{\alpha}^T \sigma \left(\mathbf{W} \left[\mathbf{h}_s^{(n)} \parallel \mathbf{h}_{i,r}^{(n-1)} \right] \right), \quad (4.8)$$

and obtain the weights of global messages

$$\alpha_s = \frac{\exp(e(\mathbf{h}_s^{(n)}, \mathbf{h}_{i,r}^{(n-1)}))}{\sum_{c \in C(i,r)} \exp(e(\mathbf{h}_c^{(n)}, \mathbf{h}_{i,r}^{(n-1)}))}. \quad (4.9)$$

It is noteworthy that the $\boldsymbol{\alpha}^T$ and $\mathbf{W}$ here are different from those of (4.4).

The aggregated global message is therefore obtained by performing a weighted sum of transformed global messages

$$\mathbf{m}_{G_{i,r}}^{(n)} = \sum_{s \in C(i,r)} \alpha_s \mathbf{W} \mathbf{h}_s^{(n)}. \quad (4.10)$$

The local node embedding is then updated based on the aggregated global message

$$\mathbf{h}_{i,r}^{(n)} = \phi_L \left(\mathbf{h}_{i,r}^{(n-1)}, \mathbf{m}_{G_{i,r}}^{(n)} \right). \quad (4.11)$$

4.3.7 The GNN Algorithm for Solving CQNs

We summarize the algorithm of our GNN approach (Algorithm 5). First, we initialize the embedding for each node in an input graph (line 1), thus there are C global node embeddings and $\sum_{r=1}^C K_r$ local node embeddings. Once node embeddings are initialized, the customized message-passing iteration is executed N times (lines 2-13). At the n -th iteration, we first aggregate local messages $\mathbf{h}_{i,r}^{(n-1)}$ to obtain $\mathbf{m}_{L_r}^{(n)}$ for each job class (line 4) and use it to update the global node embedding from $\mathbf{h}_r^{(n-1)}$ to $\mathbf{h}_r^{(n)}$ (line 5). Then, for each service center $S_{i,r}$, we aggregate associated global messages $\mathbf{h}_s^{(n)}, s \in C(i, r)$ to obtain $\mathbf{m}_{G_{i,r}}^{(n)}$ (line 9) and use it to update the local node embedding from $\mathbf{h}_{i,r}^{(n-1)}$ to $\mathbf{h}_{i,r}^{(n)}$ (line 10). After completing the customized message-passing iterations (lines 2-13), we feed global node embeddings $\mathbf{h}_r^{(N)}$ or local node embeddings $\mathbf{h}_{i,r}^{(N)}$ to two separate neural networks MLP_{global} or MLP_{local} . In this way, the GNN can make global performance predictions G_r (e.g., throughput per job class), or local performance predictions $L_{i,r}$ (e.g., response time per job class at a service center) (line

Algorithm 5 Graph neural network for closed multiclass QNs**Input:** graph representation of a closed QN with C job classes and K service centers**Output:** performance measures: $X_r, N_{i,r}, U_{i,r}, R_{i,r}$ for $r = 1, \dots, C, i = 1, \dots, K$

```

1: initialize  $\mathbf{h}_r^{(0)}, \mathbf{h}_{i,r}^{(0)}$  for  $r = 1, \dots, C, i = 1, \dots, K$ 
2: for  $n = 1, \dots, N$  do
3:   for  $r = 1, \dots, C$  do
4:     produce local message  $\mathbf{m}_{L_r}^{(n)}$  by (4.6)
5:     update global node embedding  $\mathbf{h}_r^{(n)}$  by (4.7)
6:   end for
7:   for  $r = 1, \dots, C$  do
8:     for  $i = 1, \dots, K$  do
9:       produce global message  $\mathbf{m}_{G_{i,r}}^{(n)}$  by (4.10)
10:      update local node embedding  $\mathbf{h}_{i,r}^{(n)}$  by (4.11)
11:    end for
12:  end for
13: end for
14: obtain performance measures by (4.12)

```

14).

$$G_r = MLP_{global}(\mathbf{h}_r^{(N)}), \quad L_{i,r} = MLP_{local}(\mathbf{h}_{i,r}^{(N)}). \quad (4.12)$$

The weights of our GNN are learned by minimizing the loss function defined as the mean square error (MSE) between performance predictions and ground truth. If the GNN is trained to predict global performance measures, the loss function is

$$L = \frac{1}{C} \sum_{r=1}^C (G_r - G_r^{gt})^2, \quad (4.13)$$

where G_r^{gt} represents the ground truth for global performance.

For prediction tasks on local performance measures, the GNN is trained using the loss function

$$L = \frac{1}{O} \sum_{r=1}^C \sum_{i=1}^K (L_{i,r} - L_{i,r}^{gt})^2, \quad (4.14)$$

where $O = \sum_{r=1}^C K_r$, and $L_{i,r}^{gt}$ represents the ground truth for local performance.

Table 4.2: The MAPE and percentiles of the APE distribution of queue length, utilization, response time, and throughput predictions made by AMVA, MLP, and Gsolver.

	AMVA				MLP				Gsolver			
	<i>Qlen</i>	<i>Util</i>	<i>Respt</i>	<i>Tput</i>	<i>Qlen</i>	<i>Util</i>	<i>Respt</i>	<i>Tput</i>	<i>Qlen</i>	<i>Util</i>	<i>Respt</i>	<i>Tput</i>
Mean	0.033	0.358	0.258	0.358	0.010	0.012	0.024	0.012	0.003	0.006	0.009	0.006
95th	0.119	0.700	0.564	0.700	0.012	0.040	0.053	0.040	0.007	0.020	0.029	0.020
99th	0.300	2.869	0.970	2.869	0.174	0.127	0.311	0.126	0.068	0.063	0.140	0.062

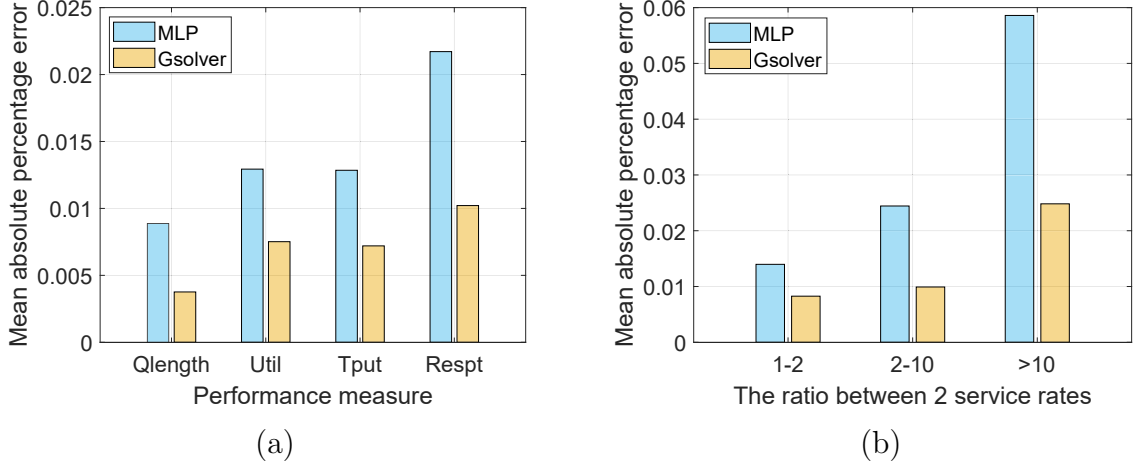


Figure 4.4: (a): MAPEs of four performance measure predictions on the second group of two-class models. (b): MAPEs of response time predictions in distinct groups classified by the difference between service times of two job classes.

4.4 Evaluation

4.4.1 Two-class Machine Repairman Networks

In this section, we first demonstrate the prediction performance of our GNN approach and MLPs using machine repairman models. As discussed in Section 4.3.1, the training and test samples of an MLP must have the same number of job classes due to its fixed input shape. Here, we focus on two-class networks and categorize them into two groups: (1) with exponential think time and service time distributions and (2) with APH think time and service time distributions.

The MLP input is a vector $\mathbf{x} = [N_1, Z_1, Z_{scv_1}, D_1, D_{scv_1}, N_2, Z_2, Z_{scv_2}, D_2, D_{scv_2}, m]$, while the output is the class-1 response time R_1 at the service center. Since there is only one service center, We omit the service center index for simplicity. According to the graph construction steps in Section 4.3.3, the input of our GNN approach is a graph with two global nodes and two

local nodes, and they have the initial features $[N_r, Z_r, Z_{scv_r}]$ and $[D_r, D_{scv_r}, m]$, respectively. The mean service time and think time range from 0.1 to 10, and their SCV range from 0.1 to 20. The population per job class is up to 50, and the number of servers takes a value from 1 to 5. The scheduling discipline of the service center is FCFS. In the first group of models, the SCV is set to 1. For each group, we train a four-layer MLP, with 64 neurons per layer, and our GNN model referred to as Gsolver using 20,000 two-class machine repairman networks, and test the MLP and Gsolver using 4,000 networks. All queueing networks in this chapter are collected using the Java Modeling Tool (JMT) simulator [BCS09]. Experiment results for the first exponential group are presented in Table 4.2, where we also add the results from the analytical method AMVA for comparison. It can be observed that both MLP and Gsolver can predict performance measures accurately, with most mean absolute percentage errors (MAPEs) around 1%, which show significantly better predictions than AMVA. The performance measure predictions for the second group of models show a similar accuracy, and their MAPEs are displayed in Figure 4.4a. Then, we classify the models into three groups based on the ratio defined as $\frac{\max(D_1, D_2)}{\min(D_1, D_2)}$ and individually visualize their response time prediction errors. An interesting observation from Figure 4.4b is that a larger disparity between the service times of different job classes poses greater challenges in prediction. The Gsolver demonstrates reliability in prediction performance as the difference grows. In summary, for fixed queueing networks, both MLPs and GNNs can learn and predict accurately. However, a distinct advantage of GNNs over MLPs is their ability to learn interdependencies and adapt to various topologies, which underscores the importance of using GNNs for performance evaluation of various queueing networks.

4.4.2 General Machine Repairman Networks

Then, we train our Gsolver using machine repairman queueing networks with a mixture of different numbers of job classes and test the Gsolver on even larger networks. We collect 24,000 queueing network samples with a non-fixed number of job classes ranging from 1 to 4, the rest of parameters are the same as those in the APH group described in Section 4.4.1. We train a Gsolver on 20,000 samples and test it using 4,000 samples as well. Figure 4.5a shows the

cumulative density functions (CDFs) of absolute percentage errors of queue length, utilization, response time, and throughput. It can be observed that over 80% of predictions have a relative error below 0.1, which indicates the effectiveness of our Gsolver in learning from flexible inputs. The MAPE of response time predictions is 0.089. We aim to further reduce this error by enhancing our Gsolver model.

Because the input features and predictions are on different scales, our first enhancement is normalizing all parameters to a range around 0 to 1. For a closed queueing network, we convert the population N_r and number of servers m_i to $\frac{1}{N_r}$ and $\frac{1}{m_i}$, respectively. We next define t_{scale} , the maximum time among all mean think times Z_r and mean service times $D_{i,r}$, that is

$$t_{scale} = \max\left(\left\{Z_r, D_{i,r} \mid r = 1, \dots, C, i = 1, \dots, K\right\}\right), \quad (4.15)$$

we then convert Z_r and $D_{i,r}$ to $\frac{Z_r}{t_{scale}}$ and $\frac{D_{i,r}}{t_{scale}}$, respectively. Finally, we convert the SCV $Zscv_r$ and $Dscv_{i,r}$ to the reciprocal of the coefficient of variation $\sqrt{Zscv_r}$ and $\sqrt{Dscv_{i,r}}$, respectively. In addition, we make a transformation for learning output. Instead of learning response time $R_{i,r}$ from the state of a local node, we learn the ratio between the mean service time and the response time, i.e., $\frac{D_{i,r}}{R_{i,r}}$, and transfer the ratio back to response time after the prediction. The steps outlined above constitute the first variant of our Gsolver model, named Gsolver- α .

In our Gsolver approach, we define global and local nodes and update their states in turns. The Gsolver flexibly predicts global or local performance measures in a queueing network based on corresponding node states. The weights of a Gsolver are trained by a loss function (4.13) or (4.14) accordingly. Based on Gsolver- α , we propose another variant Gsolver- β , which learns queueing networks under the supervision of both local and global performance measures. This is achieved by using the following loss function

$$L = \frac{1}{O} \sum_{r=1}^C \left(K_r (G_r - G_r^{gt})^2 + \sum_{i=1}^K (L_{i,r} - L_{i,r}^{gt})^2 \right), \quad (4.16)$$

where $O = \sum_{r=1}^C 2K_r$. It is noteworthy that we set the coefficient of $(G_r - G_r^{gt})^2$ as K_r to ensure equal attention on both global and local performance throughout the learning process.

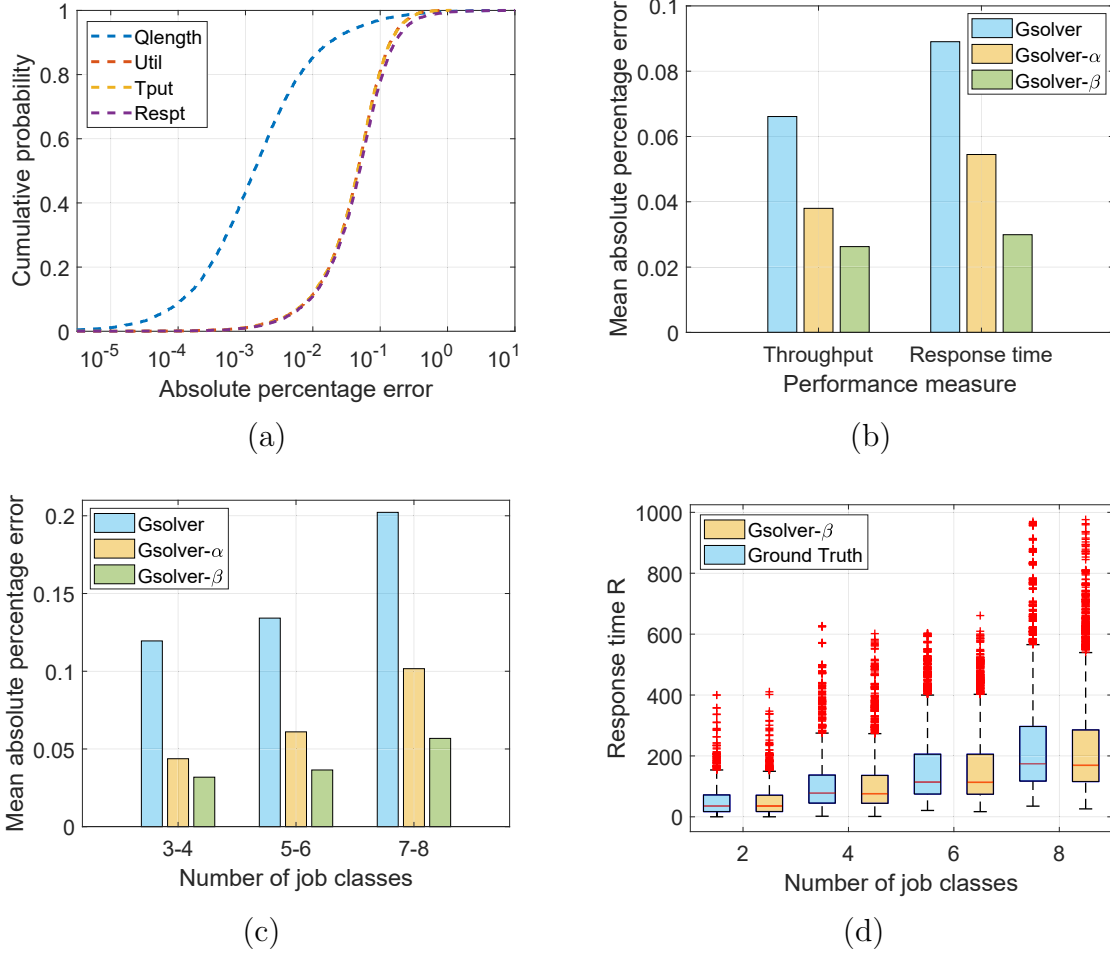


Figure 4.5: (a): CDFs of the absolute percentage errors in GNN predictions for four performance measures. (b): MAPEs of throughput and response time predictions made by the proposed Gsolver and its two variants. (c): MAPEs of response time predictions from Gsolvers across three groups, each classified by a distinct number of job classes. (d): The distributions of response time predictions and ground truth under different numbers of job classes.

Figure 4.5b shows a comparison of prediction performance among three Gsolvers. It can be observed that the proposed variants enhance the prediction effectively. For example, the MAPEs of response time predictions made by Gsolver and its two variants are 0.089, 0.055, and 0.030, respectively. Gsolver- α reduces the error by 38% compared to Gsolver, while Gsolver- β achieves a further 45% error reduction compared to Gsolver- α . Next, we evaluate the three Gsolvers on a new batch of 4,000 models, each with up to 8 job classes. Figure 4.5c displays the prediction performance when analyzing queueing networks with a rising number of job classes. It can be observed that the advantages of Gsolver- α and Gsolver- β are maintained as the number increases. Figure 4.5d visualizes the distributions of response time predictions made by Gsolver- β and ground truth, respectively. The similar distributions provided by Gsolver- β demonstrate

its effective learning and robustness gained from enhanced design.

4.4.3 Closed Multiclass Queueing Networks

In the previous section, our model training specializes in machine repairman queueing networks. In this section, we train a model using data from more general closed multiclass queueing networks, as formulated in Section 4.2. The data comes from a closed queueing network with 8 FCFS service centers. Each queueing network sample includes up to 4 job classes, with each job class travelling up to 4 service centers. Every job in the network follows a specific route determined by its class. The think times at the delay station follow an exponential distribution, while service times are from an APH distribution characterized by mean and SCV. The parameter ranges are the same as those in the APH group described in Section 4.4.1. We adopt the Gsolver- β model and learn the model parameters under the simultaneous supervision of response time (i.e., local performance measure) and throughput (i.e., global performance measure). Recall that the loss function employed by Gsolver- β is MSE, a typical loss function used for regression tasks. Here, we further explore a variant of Gsolver- β by replacing the loss function (4.16) with the following

$$L = \frac{1}{O} \sum_{r=1}^C \left(K_r \frac{|G_r - G_r^{gt}|}{G_r^{gt}} + \sum_{i=1}^K \frac{|L_{i,r} - L_{i,r}^{gt}|}{L_{i,r}^{gt}} \right), \quad (4.17)$$

and we use response time R and queue length Q as the label for local performance measure, respectively, which yields two models referred to as Gsolver- γ - R and Gsolver- γ - Q . The three models, with the identical internal architecture, are trained using 100,000 samples and tested on 20,000 unseen samples. The experiment results are shown in Figure 4.6a, where we can make two key observations. First, the loss function (4.17) enhances learning effectiveness. For example, the MAPEs in throughput predictions made by Gsolver- γ - R and Gsolver- γ - Q are around 0.011, which represent an 88.1% reduction in error compared to the 0.093 produced by Gsolver- β . Second, using queue length as the label for learning local performance measure yields better predictions than using response time. As observed, while throughput and utilization predictions from both Gsolver- γ - R and Gsolver- γ - Q have similar accuracy, Gsolver-

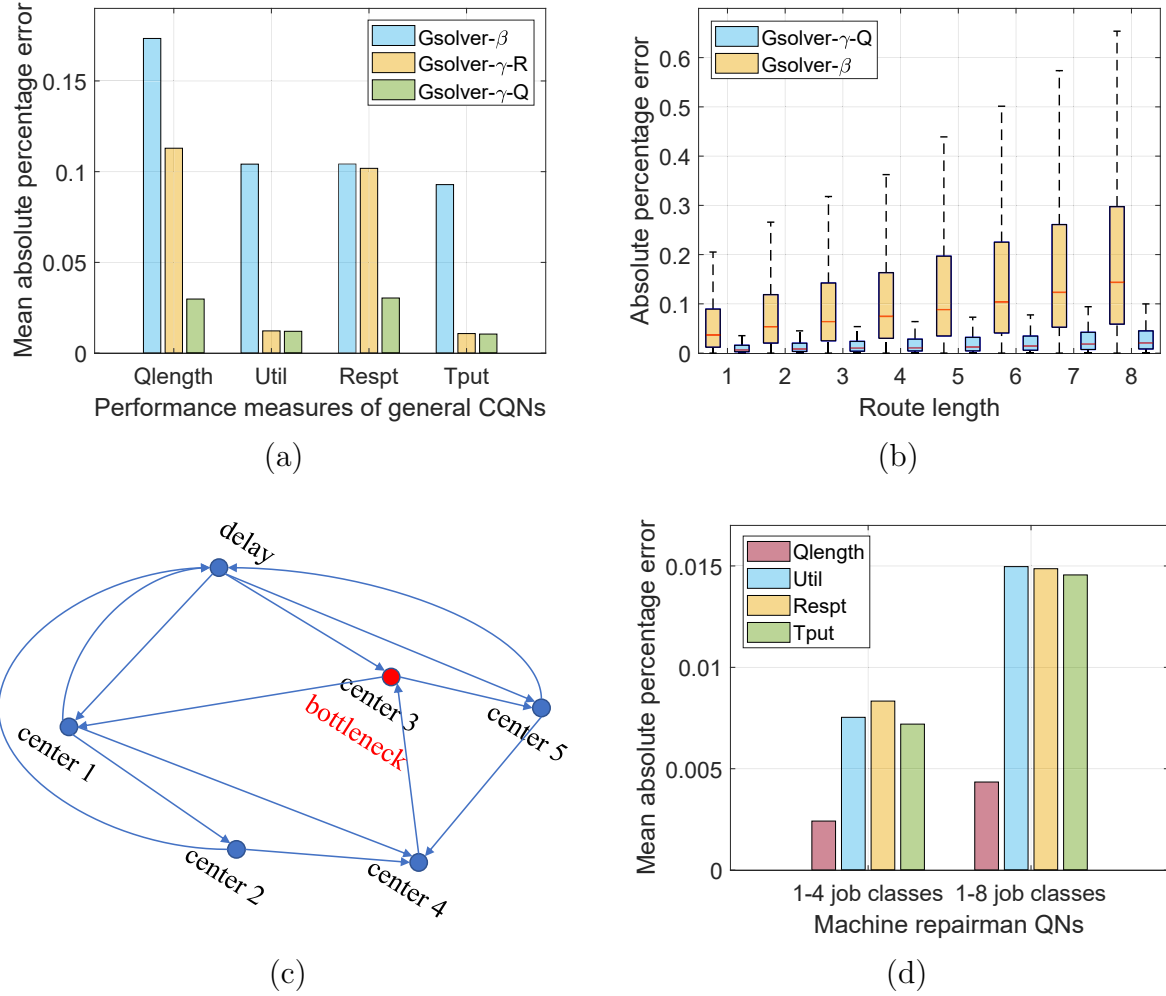


Figure 4.6: (a): MAPEs for predicting the performance measures of closed multiclass queueing networks using Gsolver- β and its two variants. (b): Box plots showing APEs for throughput predictions by Gsolver- β and Gsolver- γ -Q across different route lengths. (c): The bottleneck is the most utilized resource in the network and serves as the limiting factor for system performance. (d): MAPEs for predicting performance measures of two groups of machine repairman queueing networks using Gsolver- γ -Q.

γ -Q produces significantly lower MAPEs in queue length and response time predictions, with 0.03 compared to approximately 0.1 for Gsolver- γ -R, reducing the error by over 70%. The percentiles of APEs are displayed in Table 4.3, which reinforce the observations that learning throughput and queue length through loss function (4.17) leads to more accurate predictions.

Next, we test Gsolver- β and Gsolver- γ using a new batch of 16,000 samples. Each job class is allowed to visit up to all 8 service centers, which results in greater resource sharing and congestion in the queueing network. The box plot shown in Figure 4.6b compares throughput prediction performance of Gsolver- β and Gsolver- γ -Q across different route lengths. We can

Table 4.3: The percentiles of the APE distribution of queue length, utilization, response time, and throughput predictions made by Gsolver- β and its two variants.

	Gsolver- β				Gsolver- γ - R				Gsolver- γ - Q			
	$Qlen$	$Util$	$Respt$	$Tput$	$Qlen$	$Util$	$Respt$	$Tput$	$Qlen$	$Util$	$Respt$	$Tput$
50th	0.063	0.046	0.027	0.040	0.012	0.007	0.010	0.006	0.010	0.007	0.011	0.006
95th	0.466	0.319	0.283	0.295	0.107	0.039	0.101	0.035	0.100	0.039	0.098	0.035
99th	1.514	0.970	0.775	0.850	0.458	0.099	0.457	0.088	0.373	0.095	0.381	0.087

observe that Gsolver- γ - Q consistently produces lower APEs, and the spread of errors is much smaller than that of Gsolver- β . This demonstrates the reliability of using Gsolver- γ - Q as route length increases. We focus on throughput prediction performance, as it enables us to determine the utilization of each service center and identify the *bottleneck*. In a queueing network, as shown in Figure 4.6c, there are often one or a few service centers that are significantly more utilized than others, which are referred to as bottlenecks. Identifying the bottleneck is the key step in improving system performance, and it becomes especially challenging in networks with multiple job classes and extensive resource sharing. The utilization of a service center is equal to the sum of per-class utilizations at that service center (i.e., $U_i = \sum_{r=1}^C U_{i,r}$) [LZGS84]. For each of the 16,000 queueing network samples, we identify the service center with maximum utilization using the throughput predictions made by Gsolver- β and Gsolver- γ - Q , respectively. The bottleneck identification accuracies are 85.3% for Gsolver- β and 92.6% for Gsolver- γ - Q , which show the ability of our models to solve the bottleneck identification problem.

The Gsolver- γ - Q is trained on general closed multiclass queueing networks, which naturally include machine repairman networks. We use the two groups of machine repairman networks from Figure 4.5 to evaluate the prediction performance of the Gsolver- γ - Q . The results are shown in Figure 4.6d. As we can observe, although the training of the Gsolver- γ - Q does not exclusively focus on machine repairman networks, its predictions are highly accurate, with MAPEs of less than 0.85% and 1.51% for the two groups, respectively. Recall that the SCV of think times in these machine repairman networks ranges from 0.1 to 20. Since our Gsolver- γ - Q is trained on networks with exponential think times, we assume their think time SCV is 1 and still achieve accurate predictions. This suggests that the network performance is insensitive to the distribution of think times.

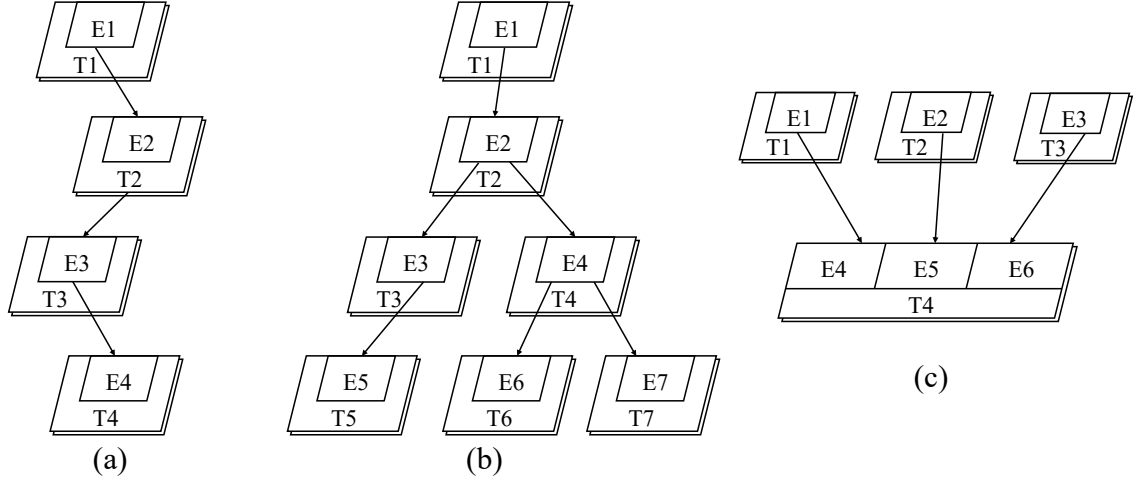


Figure 4.7: Three groups of LQNs for evaluating the prediction performance of LQN solution algorithms powered by AMVA and Gsolver, respectively.

Table 4.4: MAPEs, MdAPEs for system performance predictions made by AMVA and Gsolver and their computation times across three LQN groups shown in Figure 4.7.

LQN/Solver	LQN (a)		LQN (b)		LQN (c)	
	AMVA	Gsolver	AMVA	Gsolver	AMVA	Gsolver
MAPE (Tput)	0.229	0.040	0.269	0.097	0.633	0.056
MAPE (Respt)	0.768	0.040	2.592	0.098	0.380	0.043
MdAPE (Tput)	0.143	0.016	0.127	0.074	0.594	0.010
MdAPE (Respt)	0.177	0.015	0.136	0.068	0.397	0.010
Avg time	18.21	2.19	16.66	4.60	0.54	0.61

4.4.4 Case Studies on LQNs and Routing Probabilities

The trained Gsolver can not only solve the CQNs defined in Section 4.2 but also help in solving other problems. Here, we present two case studies: using Gsolver to solve (i) LQNs and (ii) CQNs with routing probabilities. To solve an LQN, we integrate Gsolver into the LQN solution algorithm that iteratively computes the performance of machine repairman queueing networks decomposed from the LQN by SRVN layering strategy. We conduct the evaluation using three groups of basic LQNs, as shown in Figure 4.7, with each group containing 100 LQN samples. The first group has a single chain structure with up to 10 layers and a layer multiplicity of up to 16. The second group incrementally expands the width of each layer from the first group of LQNs, forming a tree structure with up to 5 layers. The third group features multiclass clients to a common software server, with up to 4 reference tasks and a layer multiplicity of up to 5.

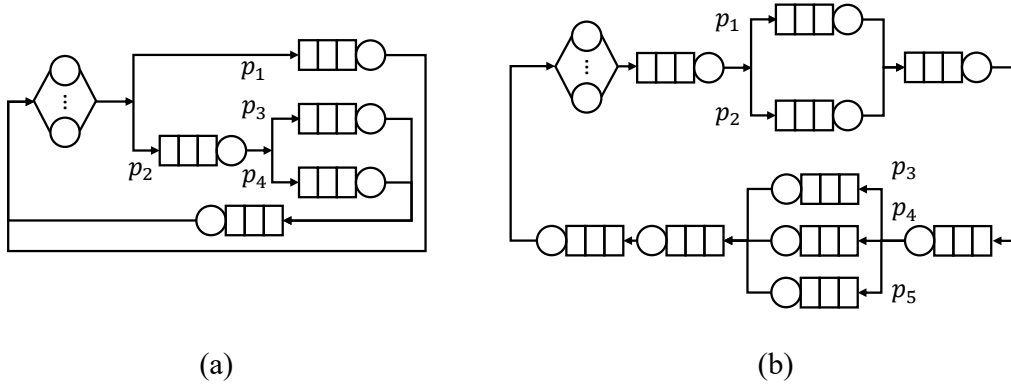


Figure 4.8: Two groups of closed queueing networks with routing probabilities for evaluating the prediction performance of AMVA and Gsolver.

The multiplicity of a reference task in these three groups ranges from 20 to 40. The service time of each entry follows an APH distribution, with the mean and SCV ranging from 0.1 to 1 and 1 to 10, respectively. Entries communicate with each other through a deterministic or stochastic call.

We compute the MAPEs and median absolute percentage errors (MdAPEs) for system response time and throughput predictions made by the Gsolver- γ - Q trained in the previous section, and compare the results with those of AMVA. As shown in Table 4.4, the iterative computation powered by Gsolver significantly reduces the errors compared to that powered by analytical method. For example, the improvement in MAPE ranges from 64% to over 90%. Additionally, using Gsolver has a lower average computation time, and tends to save more time as the number of layers increases. These experiments demonstrate the advantages of using Gsolver for solving LQN in both accuracy and efficiency.

The second additional capability of Gsolver is illustrated using CQNs with routing probabilities. Recall that Gsolver is trained on CQNs where jobs follow a linear path through service centers. To deal with CQNs with routing probabilities, we adjust the input by replacing the mean service time $D_{i,r}$ with the mean service demand $v_{i,r}D_{i,r}$, where $v_{i,r}$ is the average number of class- r job visits to the service center i , which is directly computed from routing probabilities. This is the same way of how the analytical method AMVA deals with routing probabilities. We assess the prediction performance of the trained Gsolver- γ - Q in the previous section against AMVA using

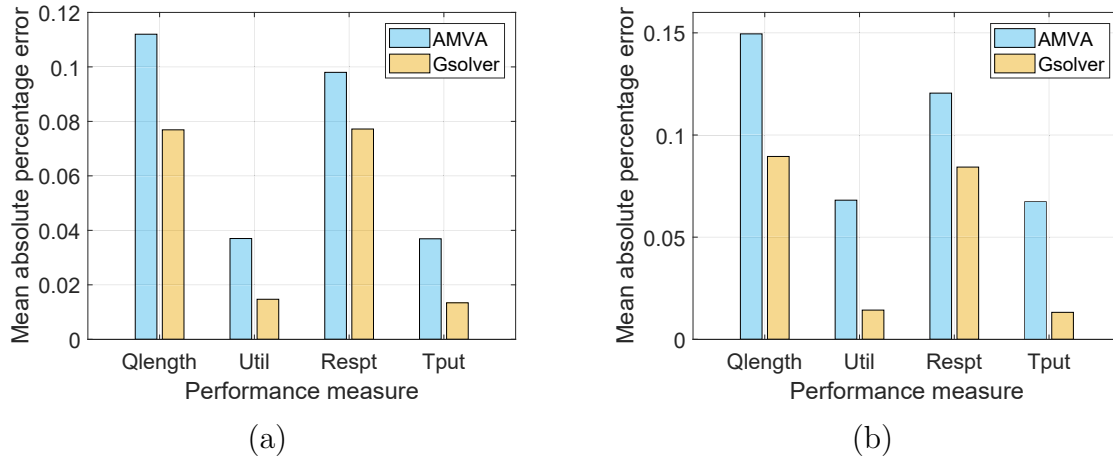


Figure 4.9: MAPEs for performance measure predictions made by AMVA and Gsolver on closed queueing networks (a) and (b) shown in Figure 4.8.

two groups of closed single-class QNs, as shown in Figure 4.8, with each group containing 100 samples. The population of each CQN, the number of servers at each service center, the mean and SCV of service time are consistent with those in Section 4.4.3. After adjusting the input, Gsolver and AMVA solve closed tandem QNs with 5 and 10 service centers. The results of the experiment are presented in Figure 4.9. It can be seen that Gsolver delivers more accurate predictions than AMVA, with the improvement becoming larger in the second group. Among the four performance measures, throughput predictions show the largest improvement, with 63.7% for the first group and 80.4% for the second group, while response time predictions see the smallest improvement, at 21.2% for the first group and 30.0% for the second group.

4.4.5 Networks with Blocking-After-Service Mechanism

So far, we have assumed that service centers in a queueing network have infinite buffers to hold waiting jobs, which means a job leaving one service center will always find space to queue at the next service center. However, in some queueing networks, service centers may have finite capacity buffers and blocking can occur when the buffer is full [BGDMT06]. Blocking-after-service (BAS) is a commonly used blocking mechanism. As illustrated in Figure 4.10, the BAS mechanism temporarily blocks a server if the buffer of the destination service center is full. The server remains blocked until a job leaves the destination service center. This creates inter-

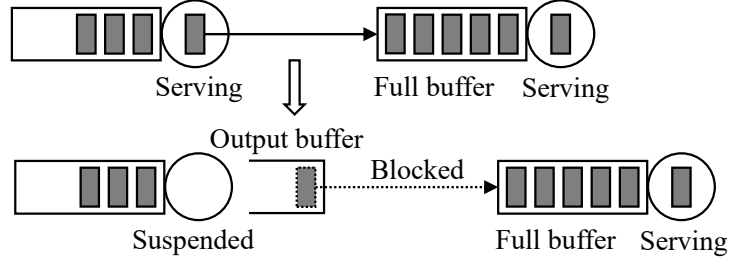


Figure 4.10: Blocking-after-service mechanism. This mechanism blocks the server when the destination reaches maximum capacity.

dependencies between service centers and leads to another non-product-form feature, making it challenging for analytical methods. In this section, we focus on closed single-class queueing networks with the BAS mechanism. The network consists of a delay station and tandem service centers, each operating under either FCFS or PS scheduling disciplines. The performance measures of interest are system response time and throughput.

We apply Gsolver to this problem, with the same architecture setting as in Section 4.4.2. Meanwhile, we use off-the-shelf GNN blocks to build a predictor, which serves as a baseline for comparison with our Gsolver. Figure 4.11 demonstrates how the baseline works. We represent the delay station and each service center with individual nodes, and connect them using directed edges that follow the service path of a job. Next, we connect undirected edges from the delay node to the intermediate service center nodes, which creates bidirectional paths that enhance message-passing efficiency. The constructed graph is then fed into a stack of graph convolution layers in which the node embeddings are iteratively updated and ultimately mapped to the system performance measure. The operations at each graph convolution layer are illustrated on the right side of Figure 4.11. For the use of Gsolver, since the goal is to predict system performance, we make predictions based on the global node state. Additionally, we incorporate one-hot encoding into the initial states of the local nodes to represent the scheduling discipline of each service center, either FCFS or PS.

We collect data from closed tandem queueing networks with 2 to 8 service centers, each having up to 5 servers. The population size ranges up to 100. Both think times and service times follow an exponential distribution with a mean varying between 0.1 and 10. We train the model on

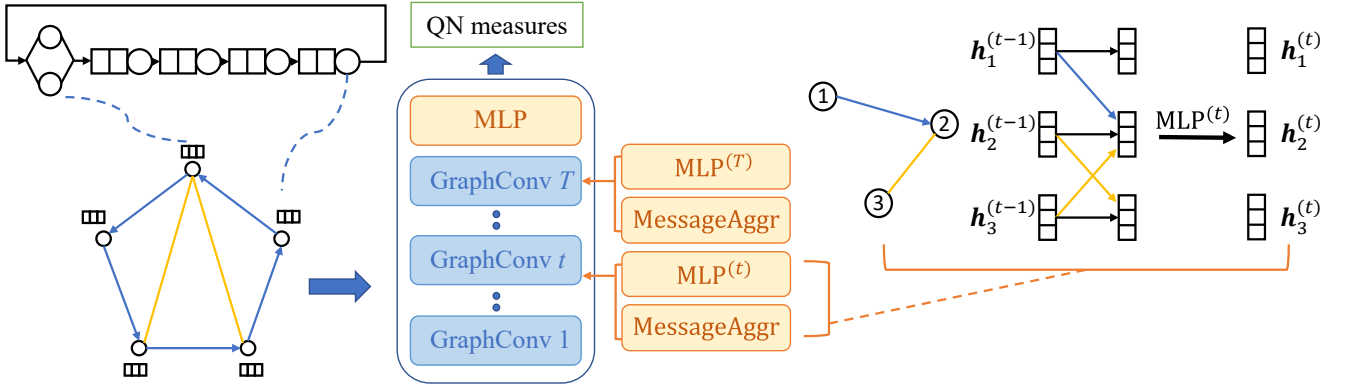


Figure 4.11: Solving closed single-class tandem QNs using off-the-shelf GNN blocks.

Table 4.5: The MAPEs and percentiles of the APE distribution of system response time and throughput predictions made by AMVA, Gsolver and baseline GNN.

	AMVA		Gsolver		GNN-baseline	
	<i>Respt</i>	<i>Tput</i>	<i>Respt</i>	<i>Tput</i>	<i>Respt</i>	<i>Tput</i>
MAPE	0.007	0.006	0.005	0.008	0.154	0.165
50th	0.002	0.001	0.003	0.004	0.101	0.122
95th	0.032	0.031	0.013	0.025	0.463	0.460
99th	0.085	0.077	0.026	0.060	0.897	0.629

10,000 samples and test it using 2,000 samples. For each queueing network sample, we simulate it under two conditions: (i) all service centers have infinite buffers, and (ii) each service center has a finite buffer with a capacity of up to 8. Therefore, we obtain two datasets. We first compare the prediction performance of AMVA, Gsolver, and baseline GNN using the dataset without finite capacity buffers. The results in Table 4.5 show that Gsolver has a similar accuracy to AMVA, with both achieving MAPEs of around 0.007 in system throughput and response time predictions. This accuracy is significantly higher than that of the baseline GNN. Next, we compare Gsolver to the baseline GNN using the dataset with finite capacity buffers and the BAS mechanism. As shown in Figure 4.12a, over 95% of the Gsolver predictions have a relative error below 0.1, while less than 50% of the baseline GNN predictions fall below 0.1. Figure 4.12a further compares the distributions of response time predictions made by Gsolver with simulation results across three groups with different network lengths. Their distribution curves show a close match. These results indicate the advantage of Gsolver for queueing problems compared to off-the-shelf GNN learning blocks.

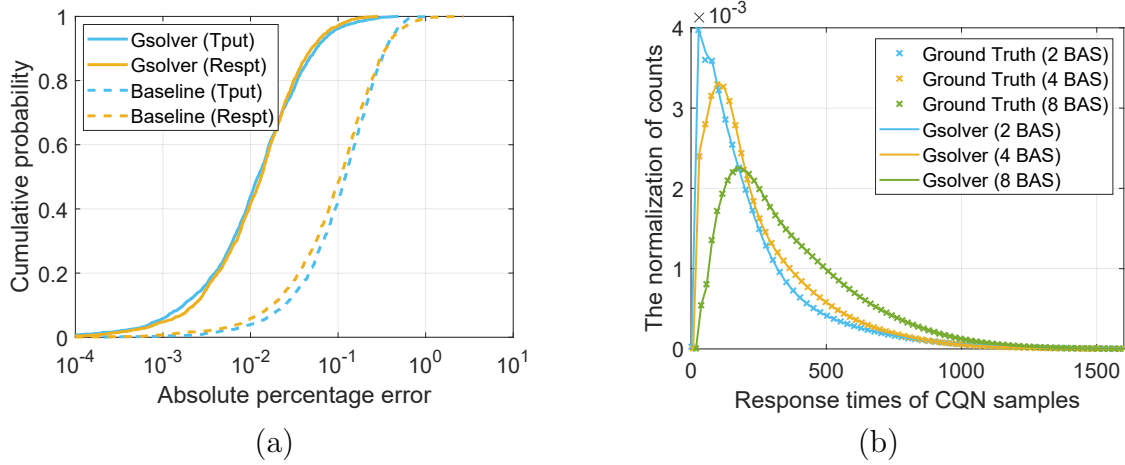


Figure 4.12: (a): CDFs of the absolute percentage errors in system throughput and response time predictions made by Gsolver and baseline GNN. (b): The distributions of response time predictions and ground truth across three groups classified by the lengths of the networks.

Table 4.6: Gsolver parameters.

	Sec 4.4.1	Sec 4.4.2	Sec 4.4.3
# hidden layer size	64	128	128
# MP iterations	2	10	12
# attention heads	4	4	12
Batch size	128	128	64

4.4.6 The configuration of model parameters

In this subsection, we provide detailed information on the Gsolver parameters used in our experiments and discuss the rationale behind their settings.

In Section 4.4.1, we begin with a simple task which is the learning of fixed two-class machine repairman networks. The used parameter values are listed in the first column of Table 4.6. Here, we investigate the impact of the number of message-passing iterations on prediction performance. The results in Table 4.7 show that 2, 4, 10, and 12 iterations yield similar performance, with 2 iterations performing slightly better.

In Section 4.4.2, we study machine repairman networks with varying numbers of job classes, a more complex task involving graphs with different sizes. We increase the hidden layer size to enhance the learning capability for this task. Here, we continue to explore how varying the number of iterations affects prediction performance. The results in Table 4.8 show that 10

Table 4.7: Absolute percentage errors in predicting two-class machine repairman networks using Gsolver, trained with different numbers of message-passing iterations.

	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
2 iter	0.003	0.007	0.006	0.020	0.009	0.029	0.006	0.020
4 iter	0.004	0.006	0.006	0.019	0.010	0.025	0.005	0.018
10 iter	0.004	0.006	0.006	0.021	0.011	0.027	0.006	0.021
12 iter	0.005	0.007	0.006	0.020	0.011	0.028	0.006	0.021

Table 4.8: Absolute percentage errors in predicting general machine repairman networks using Gsolver, trained with different numbers of message-passing iterations.

	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
2 iter	0.004	0.014	0.055	0.174	0.110	0.187	0.055	0.174
4 iter	0.003	0.012	0.052	0.149	0.080	0.158	0.052	0.150
10 iter	0.002	0.007	0.027	0.077	0.030	0.086	0.026	0.078
12 iter	0.002	0.005	0.019	0.051	0.021	0.056	0.018	0.051

and 12 iterations demonstrate substantially better performance than 2 and 4, indicating that a higher number of iterations can improve learning effectiveness on batches of non-fixed graphs.

In Section 4.4.3, we study general closed queueing networks with different topologies and varying numbers of job classes, which is the central task of this Chapter that involves capturing the complex interdependencies present in larger-scale graphs. Consequently, we increase the number of iterations to 12. Meanwhile, we increase the number of attention heads. We use a maximum of 12 in our experiments, considering the computational cost. The findings in Table 4.9 demonstrate that a greater number of iterations combined with an increased number of attention heads can lead to a notable improvement in accuracy. In addition, we adopt a smaller batch size of 64 during training, motivated by the understanding that smaller batch sizes can accelerate convergence in complex training tasks. The 64 is commonly used as a starting small size in the training [BB23].

We further investigate the effects of different numbers of message-passing iterations, hidden layer sizes, and batch sizes on the prediction performance of Gsolver for general closed queueing networks. We start with a model configuration with 12 iterations, a hidden layer size of 64, a batch size of 64, and 12 attention heads. In different training processes, we change those three

Table 4.9: Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different numbers of message-passing iterations and attention heads.

	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
4 iter, 4 heads	0.094	0.305	0.032	0.119	0.088	0.290	0.030	0.114
4 iter, 12 heads	0.061	0.214	0.027	0.096	0.060	0.207	0.025	0.091
12 iter, 12 heads	0.030	0.100	0.012	0.039	0.030	0.097	0.010	0.035

Table 4.10: Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different numbers of message-passing iterations.

	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
2 iter	0.150	0.588	0.093	0.362	0.157	0.582	0.091	0.370
4 iter	0.076	0.287	0.033	0.123	0.073	0.276	0.031	0.117
6 iter	0.061	0.231	0.024	0.086	0.061	0.225	0.022	0.082
8 iter	0.049	0.166	0.019	0.065	0.048	0.162	0.017	0.060
10 iter	0.036	0.129	0.015	0.049	0.036	0.127	0.013	0.045
12 iter	0.035	0.124	0.014	0.047	0.035	0.123	0.012	0.043
14 iter	0.039	0.142	0.017	0.057	0.040	0.142	0.015	0.053
16 iter	0.040	0.146	0.016	0.054	0.041	0.144	0.014	0.051
18 iter	0.055	0.205	0.026	0.085	0.057	0.203	0.024	0.081
26 iter	0.155	0.587	0.069	0.210	0.159	0.585	0.068	0.210
34 iter	0.203	0.795	0.081	0.264	0.201	0.791	0.078	0.261
42 iter	0.935	0.995	0.877	0.990	1.620	5.024	0.883	0.992

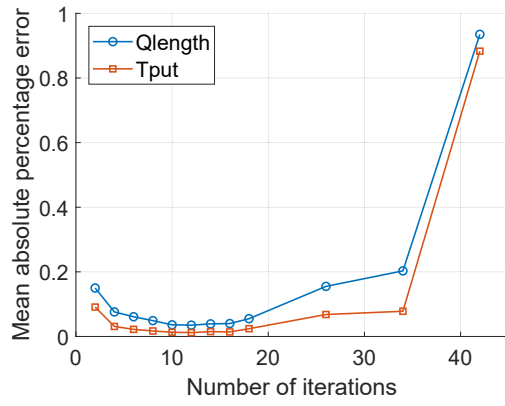
Table 4.11: Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different sizes of hidden layers.

	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
$h=16$	0.133	0.613	0.048	0.173	0.137	0.616	0.047	0.170
$h=32$	0.063	0.253	0.025	0.086	0.064	0.252	0.023	0.082
$h=64$	0.035	0.124	0.014	0.047	0.035	0.123	0.012	0.043
$h=128$	0.030	0.100	0.012	0.039	0.030	0.097	0.010	0.035
$h=256$	0.112	0.476	0.049	0.167	0.112	0.472	0.047	0.164
$h=512$	0.268	0.827	0.164	0.449	0.578	0.914	0.164	0.453

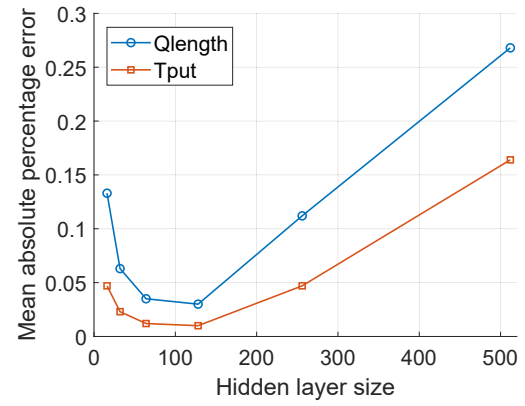
parameters within the ranges of 2-42, 16-512, and 16-1024, respectively. Then, we test the trained Gsolvers and examine the impact of distinct parameter setting. The results on varying numbers of iterations are shown in Table 4.10, the prediction performance increases first, then declines. The trend is visualized in Figure 4.13a. This could be due to a small number of iterations preventing the GNN from adequately learning the interdependencies, while a large

Table 4.12: Absolute percentage errors in predicting general closed queueing networks using Gsolver, trained with different batch sizes.

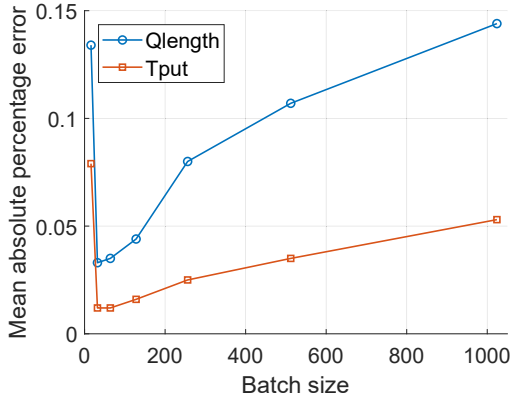
	<i>Qlen</i>		<i>Util</i>		<i>Respt</i>		<i>Tput</i>	
	MAPE	95th	MAPE	95th	MAPE	95th	MAPE	95th
$b=16$	0.134	0.490	0.081	0.227	0.162	0.506	0.079	0.223
$b=32$	0.033	0.115	0.013	0.043	0.034	0.114	0.012	0.039
$b=64$	0.035	0.124	0.014	0.047	0.035	0.123	0.012	0.043
$b=128$	0.044	0.166	0.017	0.060	0.044	0.163	0.016	0.056
$b=256$	0.080	0.339	0.027	0.097	0.081	0.337	0.025	0.092
$b=512$	0.107	0.475	0.037	0.131	0.108	0.476	0.035	0.127
$b=1024$	0.144	0.663	0.056	0.197	0.144	0.664	0.053	0.192



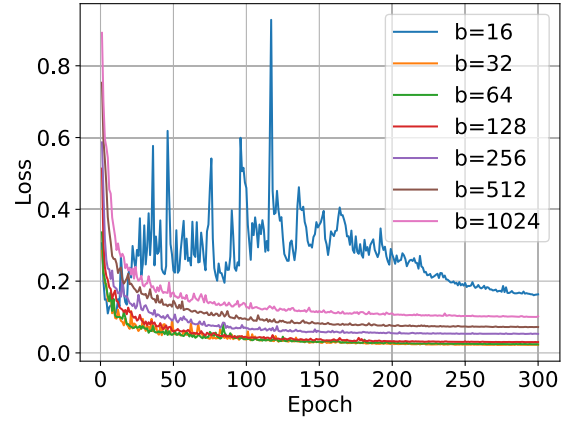
(a)



(b)



(c)



(d)

Figure 4.13: The impacts of the number of message-passing iterations, hidden layer size, and batch size on the prediction performance of Gsolver.

number of iterations tends to cause the over-smoothing problem [RBM23]. The results on varying hidden layer sizes are shown in Table 4.11 and demonstrate a similar trend as well, which is visualized in Figure 4.13b. It can be observed that 64 and 128 have the most positive

effect on the Gsolver learning process, while a large number may explode or vanish gradients and thus cause unstable training. Table 4.12 shows a decrease in prediction performance with increasing batch size from $b = 32$, visualized in Figure 4.13c, which confirms our understanding that training with a small batch of graphs typically achieves better performance within the same number of epochs. Figure 4.13d displays the validation loss for different batch sizes, we can observe that $b = 16$ causes unstable training, while the other curves demonstrate faster convergence as the batch size decreases.

All GNN models mentioned above are trained for 300 epochs with a validation split of 0.2, using the Adam optimizer with an initial learning rate of 0.001, decaying by 10% every 10 epochs.

4.5 Summary

In this chapter, we propose a GNN approach for the performance evaluation of non-product-form closed multiclass QNs with varying numbers of job classes and topologies. We summarize the main points as follows:

- The approach models a CQN using a graph with two types of nodes representing job classes and service centers, and iteratively applies message-passing between these two node types to capture their interdependencies.
- For a given CQN, our GNN model can predict throughput, response time, utilization, and queue length per job class at each service center, which reveals the potential of a learning-based model to serve as a general-purpose QN solver.
- We evaluate the prediction performance of our approach and plain regression MLPs using machine repairman QNs. We emphasize that the distinct advantage of GNNs over MLPs lies in their ability to model interdependencies and adapt to various topologies.
- We introduce a modification to the QN input features that improves the accuracy of our GNN by over 38%. Furthermore, we show that learning two performance measures simultaneously, such as throughput and response time, enhances learning effectiveness.

- We develop variants of our GNN model and evaluate them on general CQNs, where jobs follow distinct class-dependent linear routes. The best model achieves MAPEs below 0.04. Besides, we show these accurate predictions can help identify bottlenecks effectively.
- We conduct two case studies to show the additional applications of our model, which can be used to solve LQNs through integration into the iteration algorithm and deal with CQNs with routing probabilities by adjusting the input. Compared to the analytical method, the average accuracy improvement is over 50%.
- We compare our approach with off-the-shelf GNN blocks using CQNs that feature the BAS mechanism and a mixture of FCFS and PS service centers. The significantly lower error of our model demonstrates the advantage of its design tailored by queueing analysis.

Chapter 5

Customized Graph Model for Loss-aware Distributed Service Placement

5.1 Introduction

Deep neural network (DNN) models play a key role in a wide range of modern applications [LBH15, KSH12, SLJ⁺15, KT19, FC20]. Since DNN models are resource-hungry, inference jobs have been initially run only in high-performance data centers [HL22]. However, in recent years, to increase the security and privacy of users [GLME⁺15], and also to improve energy efficiency and latency [KGK⁺22], an increasingly common approach is to rely on edge architectures that run inference jobs located near the end user that generates the data [CR19]. Yet, edge devices typically have limited memory and computing resources, which limits the possibility of deploying a large DNN model on a *single* edge device. To enable deep learning at the edge, referred to in the following as edge AI, a DNN model is therefore partitioned into a set of successive fragments distributed on multiple edge devices for collaborative inference [LZZC19]. However, this architecture introduces potential reliability risks for the application, since data exchanged among edge devices may be lost due to the inability of the receiving device to accept

incoming jobs when memory is nearly exhausted. In this chapter, reliability is quantified through job loss rate, which depends on the probability of an edge device memory being full and unable to accept new jobs, consequently losing data. Therefore, a challenge exists on choosing an optimal edge deployment to minimize the data loss rate, given the knowledge of partitioned fragments and available edge devices. To answer this question, it is desirable to have an algorithmic solution to evaluate the loss rate of candidate deployments in an automated fashion, supporting the system design decisions.

Although deploying partitioned DNN models has been a topic of much research [CR19], existing research assumes that each DNN fragment will operate under a light workload with little resource contention and congestion, resulting in negligible data loss. However, this assumption is simplistic in many real scenarios. For example, a stream of images triggered by camera-detected events may occasionally have an arrival rate higher than the execution speed of the edge device. This leads to congestion and calls for queueing analysis to characterize the interplay between latency and finite memory constraints at the devices. The question is further complicated when the system accommodates multiple job arrival streams and has dependencies between the executions of DNN fragments. Thus, to address the above challenge, abstractions are needed to characterize the dependability of systems with congestion and finite memory constraints and identify an optimal deployment plan. System evaluation based on network simulation tools, such as NS-3 [RH10], may provide a reasonable approximation of the real system behavior, but they are typically too slow to be used in complex optimization programs, where hundreds or thousands of models need to be evaluated while seeking for a globally optimal decision. Stochastic models, such as queueing networks (QNs), offer more abstract approximations, which trade accuracy of the system representation for speed [BGDMT06]. Yet, explicit solutions in the presence of multiple service chains and finite constraints are not known, due to the loss of product-form characteristics [Tho06]. Simulation can still be adopted also for these models but, while faster than system simulation, it still incurs a computational bottleneck within optimization programs.

To address this issue, we focus on graph representation learning that has recently emerged as a way to learn and evaluate complex networks [HYL17]. A learned graph neural network

(GNN) can capture complicated dependencies within graphs unseen in the training phase. Besides, GNN inference is more scalable than queueing simulations and thus can be used to evaluate large distributed networks. Motivated by the difficulty in analyzing data loss and the accuracy of graph representation learning, we propose in this chapter a novel family of GNN models, called ChainNet, that can efficiently and accurately assess the reliability of deployments facing resource contention and data loss. We capture the underlying queueing structure of an edge AI system through ChainNet. Compared to popular off-the-shelf GNN models, ChainNet internally adopts a customized convolution strategy that accounts for the inter-dependencies among common system metrics. ChainNet has the potential to serve as a general framework for the GNN modeling of DNN deployments, which supports decision-making in the deployment design. Although ChainNet is employed in this study to distribute DNN fragments, its applicability extends in principle to similar problems for other distributed systems. Its main assumption is that the studied service consists of a linear chain of fragments so that the routing is deterministic, which holds true for most distributed DNNs.

The rest of the chapter is organized as follows: the loss-aware deployment problem is formulated in Section 5.2. A motivating example is presented in Section 5.3. ChainNet and its generalized design are developed in Section 5.4 and Section 5.5. The surrogate optimization program is illustrated in Section 5.6. Evaluation results are shown in Section 5.7 and the conclusion is drawn in Section 5.8.

5.2 Problem Formulation

An edge AI service system comprises a set of D heterogeneous devices indexed by $k = 1, \dots, D$. We assume these devices to be able to acquire data and trigger processing at other devices. We assume that links exist between any two devices offering a stable transmission path (e.g., WiFi or Bluetooth) with negligible signal degradation (e.g., clear line-of-sight). Additionally, we assume all devices are located within the transmission range of the chosen network technology. The system is populated by a set of C distinct service chains indexed by $i = 1, \dots, C$. Each

such chain represents an AI application executed on the edge infrastructure. A chain- i service consists of a chain of DNN fragments indexed by $j = 1, \dots, T_i$, which are executed in sequential order. Each of its fragments is executed on a separate device.

We assume the arrival of chain- i service requests follows a Poisson process of rate λ_i . A fragment j is characterized by its memory demand $m_{i,j}$ and computational demand $r_{i,j}$. A device k is characterized by its maximum memory capacity M_k and a service rate R_k . An incoming job is either queued in the buffer of k or dropped if available memory is insufficient to accept another request. A device k can be shared by fragments of multiple service chains. All devices use First-Come-First-Served (FCFS) scheduling to execute queued jobs. The processing time of a fragment j at a device k is defined as the ratio between $r_{i,j}$ and R_k .

A placement decision $p \in \mathcal{P}$, where $\mathcal{P}$ is the space of all possible placements, assigns the fragments to the available devices, and it is described by variables

$$p_{i,j,k} = \begin{cases} 1 & \text{if device } k \text{ executes fragment } j \text{ of service } i \\ 0 & \text{otherwise} \end{cases}. \quad (5.1)$$

Thus, a placement p is determined by $D \sum_{i=1}^C T_i$ variables $p_{i,j,k}$. A placement may not use all of the available devices, thus we let d denote the number of used devices in a placement, $d \leq D$.

Predicting the performance of a given placement is important as it enables us to allocate resources in an optimal manner. Performance measures that are commonly of interest include system throughput and end-to-end latency per service chain. The inference accuracy is not a figure of merit for us since we assume that the DNN models are already trained and we neither vary the model architecture nor the weights. The system throughput $X_i \equiv X_i(p)$ is defined as the mean number of chain- i requests completed in a single unit of time. The end-to-end latency L_i is composed of two parts: (i) total latency of fragments $j = 1, \dots, T_i$ at the devices and (ii) total transmission time between the devices. The total latency of fragments is made up of total processing time of fragments and total queueing time at the devices. The problem addressed is the optimal placement to minimize the overall data loss rate of C service chains,

in other words, to maximize the total system throughput, i.e.,

$$\begin{aligned} \max_{p \in \mathcal{P}} \quad & X_{total}(p) = \sum_{i=1}^C X_i(p) \\ \text{s.t.} \quad & \Delta m_k \leq M_k, \quad \forall k \in \{1, \dots, D\}, \end{aligned} \tag{5.2}$$

where Δm_k is the sum of memory requirements of all types of fragments executed on the device k , i.e., $\Delta m_k = \sum_{i=1}^C \sum_{j=1}^{T_i} p_{i,j,k} m_{i,j}$. This constraint is meant to consider the limit on memory capacity to ensure a feasible deployment at each edge device.

5.3 A Motivating Example

Queueing network (QN) models are a widely used performance modeling tool for AI and edge systems [LHAES23, SLA⁺22, DXT⁺20]. Since service requests received by edge devices are external arrivals, edge AI service systems may be modeled as *open* QNs. In particular, we use queueing networks to model the key aspects behind loss and contention that arise in such devices. An open QN has external arrivals and consists of a set of queueing stations. Each edge device is modeled as a station. Queues build up whenever arrival requests find servers are busy. Each station has a finite-capacity buffer to store the queue of incoming jobs. Any arrival that finds no room is lost due to buffer overflow. Stations are connected by service routes that are fixed once the placement decision is enacted.

Abstracting the system as an ideal open QN, it is possible to show that the system throughput and the queueing time at individual devices are unaffected by the network transmission time since the latter acts as a pure delay. In this chapter, consistently with this observation, we do not explicitly model the total network transmission time, only the performance at the devices. Figure 5.1 provides an example of a placement decision. The fragments of three service chains are executed by four devices. The arrows represent service routes; a chain- i arrow from device k to k' depicts that a chain- i request that finishes service at device k is next routed to device k' . This placement is modeled as the open QN shown in Figure 5.2.

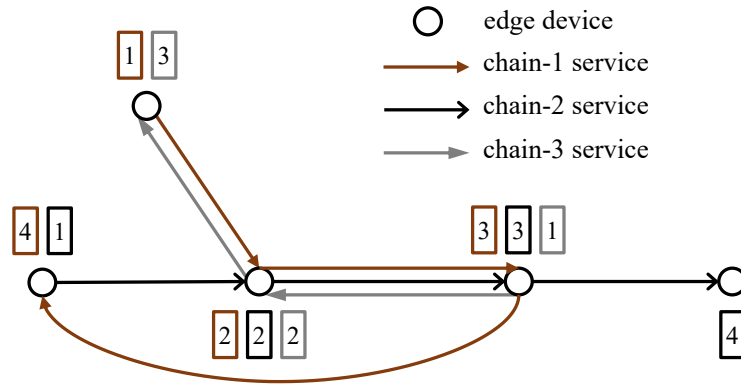


Figure 5.1: An example of deploying three edge AI services on five devices. Both chain-1 and chain-2 services comprise four fragments, while chain-3 service comprises three fragments. Fragments are represented by boxes. The number within a box represents the position of this fragment in its service chain.

The analysis of open QNs with losses is however challenging. In particular, obtaining exact closed-form solutions for the steady-state queue-length distributions is generally not feasible. In [BS17], the authors achieve the exact analysis by considering a small two queue tandem network in which losses only occur at the first queue. While some works have undertaken approximate analysis for non-product form networks with finite capacity and losses [Shi95, CPR19], they focus only on single-chain settings, while here we consider a multi-chain setting for which, to the best of our knowledge, accurate approximations do not exist in the literature. In particular, our models are rooted in the needs of edge AI systems, where the loss process is driven by memory constraints. Such constraints are known to have limited tractability, as they are known to yield product-form solutions only if the loss occurs at elements without buffers [KM02]. Therefore, we explore building a learning-based surrogate model for the reliability assessment of candidate large-scale multi-chain placement decisions.

5.4 Methodology

5.4.1 Workflow

Figure 5.3 shows the proposed optimization workflow. Given the available devices and edge AI services to be deployed, our optimizer can return the best placement decision with respect to

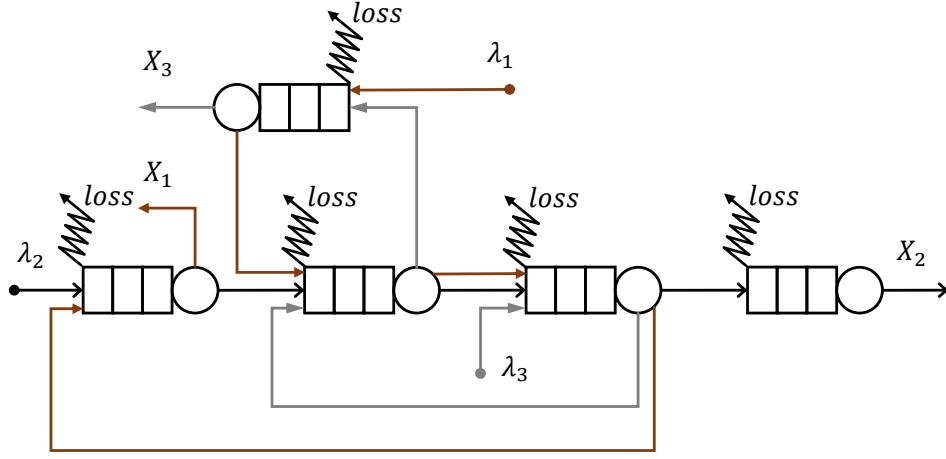


Figure 5.2: The queueing network model of the placement in Figure 5.1. A station with a finite-capacity buffer is represented by a circle followed by a series of rectangles. The arrival rates of requests for three services are λ_1 , λ_2 , and λ_3 .

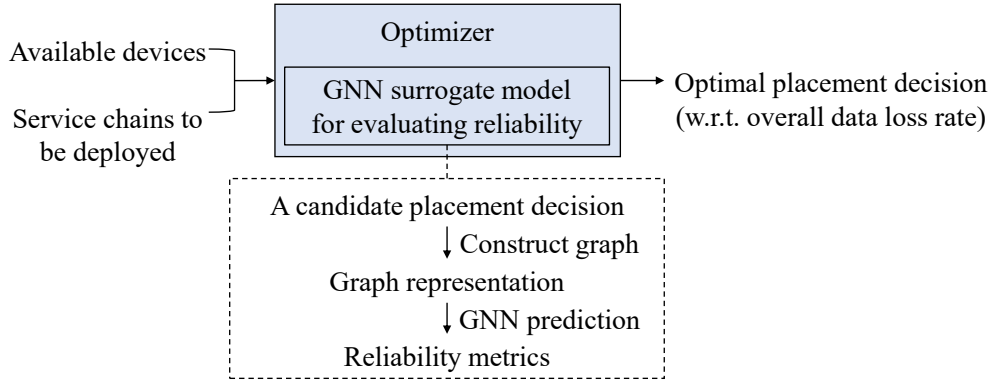


Figure 5.3: Main steps of the proposed approach.

overall system throughput according to (5.2). The central part of the optimizer is ChainNet, serving as a surrogate to estimate the performance of candidate placement decisions.

5.4.2 Model input: graph representation and features

To assess the reliability of a placement decision p , we first represent it as a heterogeneous graph where each node or edge corresponds to a component or a dependency involved in p (Algorithm 6). It serves as the input to our GNN model. A placement decision p involves components including services, fragments, and devices. We thus create three types of nodes to represent them respectively (line 1). As a result, the graph has $C + \sum_{i=1}^C T_i + d$ nodes in total. Besides, there exist two relationships: (i) the placement of a fragment and (ii) the execution

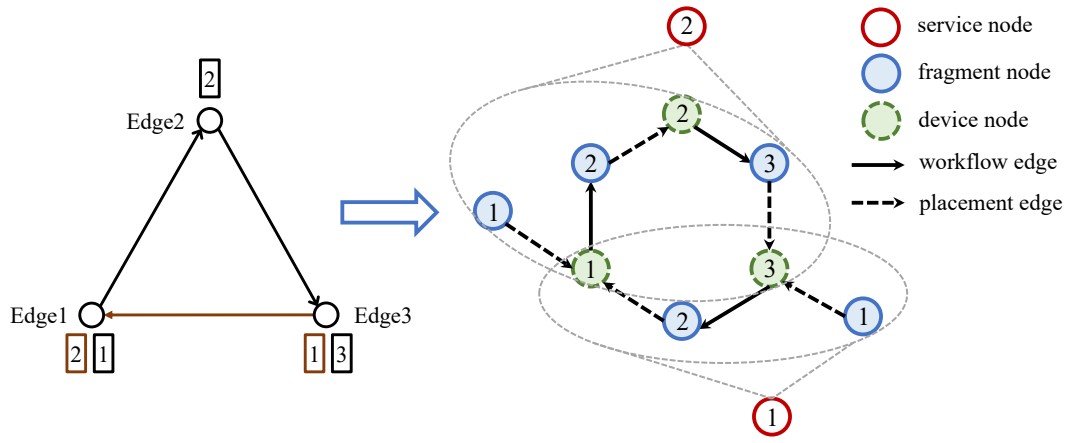


Figure 5.4: Creating input graph corresponding to the placement decision. There are five types of elements in a graph representation: (i) service nodes: hollow circles with solid red borders; (ii) fragment nodes: filled nodes with solid blue borders; (iii) device nodes: filled nodes with dashed green borders (iv) workflow edges: solid arrows; (v) placement edges: dashed arrows.

order of sequential fragments, which we capture with two types of edges. The *placement edges* point from every fragment node to its device node (lines 2-4), while *workflow edges* point from the device node of a fragment node to its subsequent fragment node except for the final fragment (lines 5-7). Service nodes are not directly connected to any other nodes in the graph. They serve as a hypernode that tracks associated fragment nodes and device nodes.

The feature of a service node v_i is the arrival rate λ_i . For a fragment node v_j , the features include the processing time of this fragment at its device, $\frac{r_{i,j}}{R_k}$, and the memory demand of this fragment, $m_{i,j}$. The feature of a device node v_k is its maximum memory capacity M_k . The details of initial node embeddings are given later in Table 5.2.

An example of graph construction is displayed in Figure 5.4. This decision uses three edge devices to execute two chains of services that comprise two and three fragments, respectively. We create a total of ten nodes, and then use workflow edges and placement edges to connect all non-service nodes.

5.4.3 Design rationale based on queueing analysis

Graph partition

We partition the entire graph into a set of *execution steps*. An execution step is comprised of

Algorithm 6 Graph construction**Input:** a placement decision p **Output:** a graph representation

- 1: create service, fragment, and devices nodes: v_i, v_j, v_k
- 2: **for** each non-zero $\alpha_{i,j,k} \in p$ **do**
- 3: connect $v_j \rightarrow v_k$
- 4: **end for**
- 5: **for** each non-zero $\alpha_{i,j \neq T_i,k} \in p$ **do**
- 6: connect $v_k \rightarrow v_{j'}$ (j' is the subsequent fragment of j)
- 7: **end for**

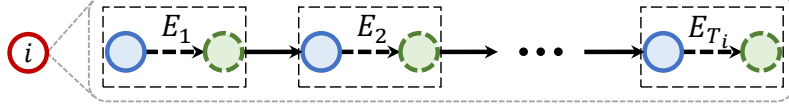


Figure 5.5: The execution sequence of the chain- i service comprising T_i fragments. It is partitioned into execution steps $E_1, \dots, E_{T_i}$. The color scheme follows the one used in Figure 5.4.

a fragment node v_j , a device node v_k , and a placement edge from v_j to v_k . Each service node is associated with a set of execution steps connected by workflow edges, which compose its *execution sequence*. For instance, as shown in Figure 5.5, the execution sequence of the chain- i service is $E_1 \rightarrow \dots \rightarrow E_{T_i}$, where $E_j, j = 1, \dots, T_i$ are execution steps.

Therefore, a graph modeling C service chains has C execution sequences and $\sum_{i=1}^C T_i$ execution steps. It is noteworthy that a fragment node is unique to its execution step, while a device node might be shared by multiple execution steps.

Latent representations of performance measures

The end-to-end latency of a service chain is the sum of the latencies of its fragments. In the proposed GNN model, we use the sum of the fragment node embeddings as the latent representation of the end-to-end latency.

Because arrivals get dropped when the buffer is full, throughput typically shows a gradual decrease along the execution sequence. As a result, system throughput equals the throughput at the last execution step of the execution sequence. Following the execution sequence, we recurrently update the embedding of the service node at every execution step. The embedding updated at the last execution step acts as the latent representation of the system throughput.

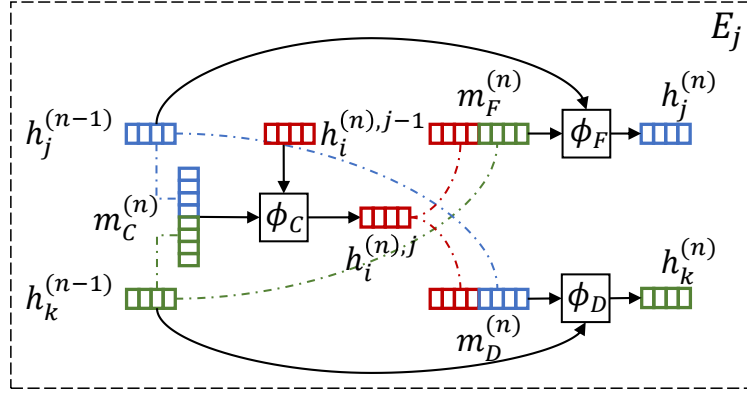


Figure 5.6: The n -th iteration of the proposed message passing within the j -th execution step of the chain- i service. The color scheme follows the one used in Figure 5.4. The colored dash-dotted lines copy node embeddings. The black arrows indicate input/output of the three update functions ϕ_C , ϕ_F , and ϕ_D .

Identifying dependency within one execution step

Queues at devices are stable due to their finite buffers. In this case, every execution step E_j can show a steady-state behavior where the (i) average latency L_{E_j} , (ii) average throughput X_{E_j} , and (iii) average queue length Q_{E_j} remain in equilibrium over time. The queue length indicates the extent of resource contention at a device. The state of a device is thus captured by the embedding of the device node.

According to Little's law [JC61], the measures (i)-(iii) are interdependent under steady-state conditions. We leverage this mutual dependency in the design of message aggregation within each execution step by relating messages in patterns that resemble the interdependencies that the measures have among themselves.

5.4.4 Model design for a single service chain

The reason we partition the execution sequence of a service chain into a set of execution steps is that the latter is the basic unit used to perform message passing. As shown in Figure 5.6, three types of embeddings are involved in this process, which we now discuss in the next subsections.

Service node embedding

The service node is a hypernode tracking the associated execution sequence, its representation

learning needs to take into account the order of execution steps and their inter-dependencies. At each iteration, we recurrently update the service node embedding at execution steps E_j , $j = 1, \dots, T_i$ using an update function ϕ_C

$$\mathbf{h}_i^{(n),j} = \phi_C\left(\mathbf{h}_i^{(n),j-1}, \mathbf{m}_C^{(n)}\right), \quad (5.3)$$

where $\mathbf{m}_C^{(n)}$ is the service message. The service node embedding is updated T_i times per iteration. The final embedding from the previous iteration serves as the initial embedding for the next iteration, i.e.,

$$\mathbf{h}_i^{(n+1),0} = \mathbf{h}_i^{(n),T_i}. \quad (5.4)$$

The final embedding at the last iteration is used as the latent representation of system throughput of the chain- i service.

Regarding the service message $\mathbf{m}_C^{(n)}$, the mutual dependency guides us to aggregate the embeddings of the fragment node and device node within the same execution step to produce the message

$$\mathbf{m}_C^{(n)} = f\left(\left\{\mathbf{h}_j^{(n-1)}, \mathbf{h}_k^{(n-1)}\right\}\right). \quad (5.5)$$

Fragment node embedding

The fragment node embedding is related to the latency of this fragment, which is updated once per iteration

$$\mathbf{h}_j^{(n)} = \phi_F\left(\mathbf{h}_j^{(n-1)}, \mathbf{m}_F^{(n)}\right). \quad (5.6)$$

The fragment message $\mathbf{m}_F^{(n)}$ is built using the service node embedding from the same execution step as well as the device node embedding

$$\mathbf{m}_F^{(n)} = f\left(\left\{\mathbf{h}_i^{(n),j}, \mathbf{h}_k^{(n-1)}\right\}\right). \quad (5.7)$$

Device node embedding

The state of the device is captured by its node embedding, which is updated once per iteration

as well

$$\mathbf{h}_k^{(n)} = \phi_D\left(\mathbf{h}_k^{(n-1)}, \mathbf{m}_D^{(n)}\right). \quad (5.8)$$

Similarly, the device message $\mathbf{m}_D^{(n)}$ is built by aggregating the node embeddings of the two other types from the same execution step

$$\mathbf{m}_D^{(n)} = f\left(\left\{\mathbf{h}_i^{(n),j}, \mathbf{h}_j^{(n-1)}\right\}\right). \quad (5.9)$$

Message aggregation and update

In our framework, $\mathbf{m}_C^{(n)}$, $\mathbf{m}_F^{(n)}$, and $\mathbf{m}_D^{(n)}$ are aggregated messages, returned by function f , that are used to update the embeddings of service, fragment, and device nodes, respectively. Because an aggregated message is built using node embeddings from two different types, we concatenate the two embeddings to obtain the aggregated message

$$f(\{\mathbf{x}, \mathbf{y}\}) = [\mathbf{x} \parallel \mathbf{y}], \quad (5.10)$$

where $\parallel$ is the concatenation operation.

In an execution step, the learning part of our GNN model comprises three neural networks, ϕ_C , ϕ_F , and ϕ_D , serving as update functions for the three types of nodes. We use three separate GRU cells [CVMG⁺14] as the three update functions. The weights of each GRU cell are shared across all execution steps.

5.4.5 Overall Algorithm

Based on previous developments, we summarize the algorithm of our GNN model (Algorithm 7).

First, we initialize the embeddings of three types of nodes (line 1). Once node embeddings are initialized, the customized message-passing iteration is executed N times (lines 2-16). At each iteration, we track the execution sequence of each service chain (lines 3-11). This is done by propagating messages from the first execution step to the last (lines 4-9). In an execution step,

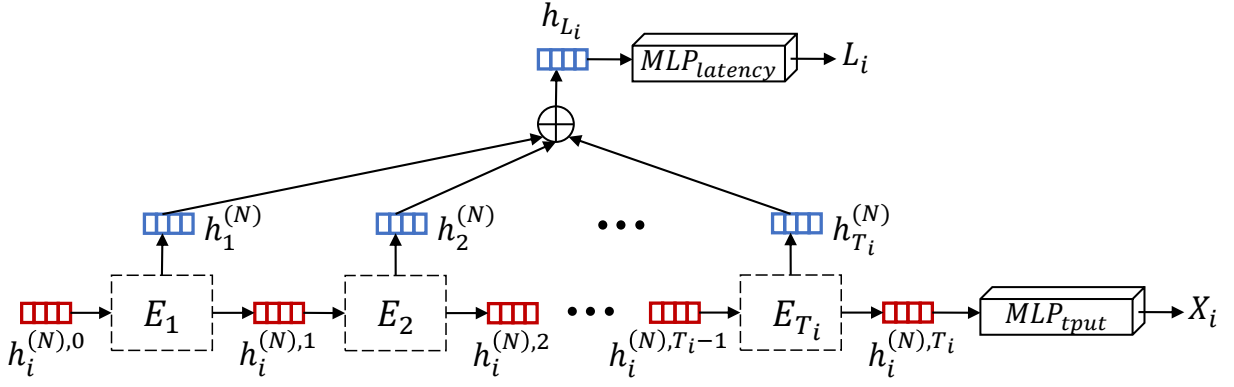


Figure 5.7: Predicting system throughput and end-to-end latency of the chain- i service concurrently at the last (N -th) iteration. The $\oplus$ represents element-wise addition of vectors. The color scheme follows the one used in Figure 5.4.

Table 5.1: Main notation used by ChainNet.

i	service chain index
j	fragment index
k	device index
n	iteration index
C	the number of service chains to be deployed
T_i	the number of fragments of the chain- i service
D	the number of available devices
F_k	the number of execution steps that include device k
p	a placement decision
d	the number of devices used by p
f	message aggregation function
ϕ	message update function
$\mathbf{h}_i$	service node embedding
$\mathbf{h}_j$	fragment node embedding
$\mathbf{h}_k$	device node embedding

we update the embeddings of the service node and fragment node (lines 5-8). After completing message-passing along the entire execution sequence, we update the initial embedding of the service node for use in the next iteration (line 10). Then, we update the node embeddings of used devices (lines 12-15).

At the end of the last iteration, we collect the embeddings of the service node, $\mathbf{h}_i^{(N),T_i}$, and all fragment nodes, $\mathbf{h}_1^{(N)}, \dots, \mathbf{h}_{T_i}^{(N)}$, of every service chain. Following the design rationale in Section 5.4.3, for each service chain, we extract the latent representations of its throughput and latency. We then feed them into two separate neural networks MLP_{tput} and $MLP_{latency}$, respectively. In this way, the system throughput and end-to-end latency are predicted concurrently

Algorithm 7 Predicting performance measures of a placement**Input:** graph representation of a placement**Output:** performance measures: X_i and L_i for $i = 1, \dots, C$

```

1: initialize  $\mathbf{h}_i^{(1),0}, \mathbf{h}_j^{(0)}, \mathbf{h}_k^{(0)}$ 
2: for  $n = 1, \dots, N$  do
3:   for  $i = 1, \dots, C$  do
4:     for  $j = 1, \dots, T_i$  do
5:       produce service message  $\mathbf{m}_C^{(n)}$  by (5.5)
6:       update service node embedding  $\mathbf{h}_i^{(n),j}$  by (5.3)
7:       produce fragment message  $\mathbf{m}_F^{(n)}$  by (5.7)
8:       update fragment node embedding  $\mathbf{h}_j^{(n)}$  by (5.6)
9:     end for
10:    initialize  $\mathbf{h}_i^{(n+1),0}$  for the next iteration by (5.4)
11:  end for
12:  for  $k = 1, \dots, d$  do
13:    produce device message  $\mathbf{m}_D^{(n)}$  by (5.9)
14:    update device node embedding  $\mathbf{h}_k^{(n)}$  by (5.8)
15:  end for
16: end for
17: obtain performance measures by (5.11)

```

(line 17)

$$X_i = MLP_{tput}(\mathbf{h}_i^{(N),T_i}), \quad L_i = MLP_{latency}(\mathbf{h}_{L_i}), \quad (5.11)$$

where $\mathbf{h}_{L_i} = \sum_{j=1}^{T_i} \mathbf{h}_j^{(N)}$. This mechanism of prediction after completing the last (N -th) iteration is illustrated in Figure 5.7.

For the training phase of our GNN model, the weights of five neural networks ($\phi_C, \phi_F, \phi_D, MLP_{tput}$ and $MLP_{latency}$) are initialized via the Glorot approach [GB10]. This method keeps the variance of activation values and gradients roughly constant throughout the forward and backward passes, which helps to avoid the exploding or vanishing gradient problem in the training process. The weights are learned by minimizing the mean square error (MSE) between the predicted and ground truth performance measures

$$L = \frac{1}{2Q} \sum_{i=1}^Q \left((X_i - X_i^{gt})^2 + (L_i - L_i^{gt})^2 \right), \quad (5.12)$$

where $Q = \sum_{g=1}^G C_g$, G is the number of graphs in a training batch, C_g is the number of service chains in the g -th graph, X_i^{gt} and L_i^{gt} are the ground truth.

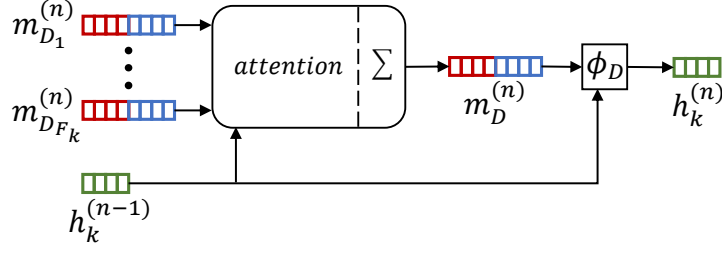


Figure 5.8: Message passing for a device node shared by multiple execution steps. The color scheme follows the one used in Figure 5.4.

5.5 Generalized GNN for Larger Placements

5.5.1 Design for a shared device by multiple execution steps

A device could be allocated to execute multiple fragments of distinct service chains, which means it could be shared by multiple execution steps. Suppose a device k is included by multiple execution steps indexed by $t = 1, \dots, F_k$, we aggregate device messages of these execution steps to obtain an overall $\mathbf{m}_D^{(n)}$

$$\mathbf{m}_D^{(n)} = f_{\text{multi}}\left(\left\{\mathbf{m}_{D_1}^{(n)}, \dots, \mathbf{m}_{D_{F_k}}^{(n)}\right\}\right). \quad (5.13)$$

Then, we use the aggregated message $\mathbf{m}_D^{(n)}$ to update the state of the shared device in the same way as (5.8).

For each involved execution step $t = 1, \dots, F_k$, we obtain its device message $\mathbf{m}_{D_t}^{(n)}$ using (5.9).

Next, we adopt a graph attention mechanism [BAY21] to compute the attention score that characterizes the significance of a device message to the shared device

$$e(\mathbf{h}_k^{(n-1)}, \mathbf{m}_{D_t}^{(n)}) = \boldsymbol{\alpha}^T \sigma\left(\mathbf{W} \left[\mathbf{h}_k^{(n-1)} \parallel \mathbf{m}_{D_t}^{(n)}\right]\right), \quad (5.14)$$

where $\mathbf{W}$ is a matrix of learnable weights, σ is a LeakyReLU activation function, and $\boldsymbol{\alpha}^T$ is the transpose of a vector of learnable weights.

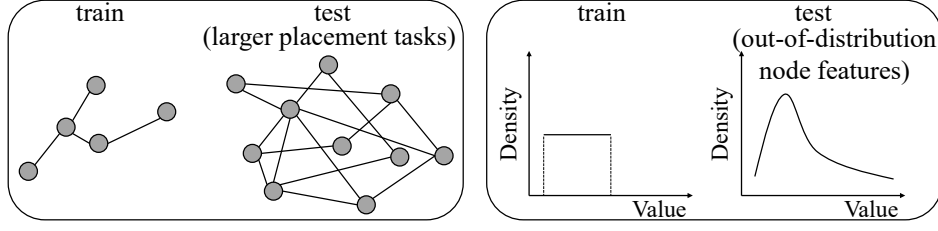


Figure 5.9: Two out-of-distribution scenarios considered by our surrogate model.

We then normalize attention scores of all F_k device messages using the Softmax function

$$\alpha_{kt} = \frac{\exp(e(\mathbf{h}_k^{(n-1)}, \mathbf{m}_{D_t}^{(n)}))}{\sum_{u=1}^{F_k} \exp(e(\mathbf{h}_k^{(n-1)}, \mathbf{m}_{D_u}^{(n)}))}. \quad (5.15)$$

Using the normalized attention, the aggregated device message is finally obtained as a weighted sum of transformed device messages given by

$$f_{multi} = \sum_{t=1}^{F_k} \alpha_{kt} \mathbf{W} \mathbf{m}_{D_t}^{(n)}. \quad (5.16)$$

In summary, to adapt our GNN to multi-chain service scenarios, we adjust the proposed message-passing process for device nodes by producing an overall device message based on the attention mechanism so that the impact of different execution steps on the shared device can be considered. This adjustment is shown in Figure 5.8.

5.5.2 Design for generalization in out-of-distribution scenarios

In real-world conditions, there will be scenarios where distribution shifts occur between training and test graphs. As shown in Figure 5.9, the out-of-distribution scenarios can occur in both network size and node features. The goal is to ensure that our GNN model attains generalization performance when it confronts such distribution shifts.

Larger placement problems

The proposed GNN model needs to generalize well to larger placement problems beyond training samples. Compared to the training samples, they could have more service chains, more fragments within a service chain, or more available devices.

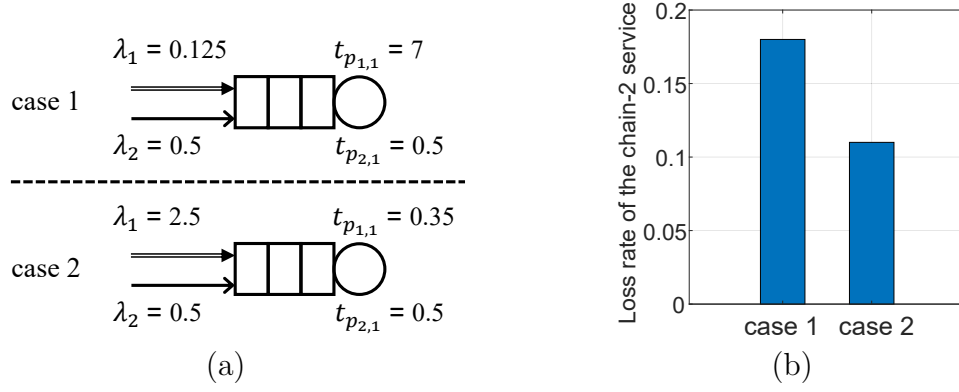


Figure 5.10: (a) Suppose the only differences between these two cases are the arrival rate and processing time of the chain-1 service. The modified features of the processing time in the two cases are the same: $t_{p_{1,1}}\lambda_1 = 0.875$ and $t_{p_{2,1}}\lambda_2 = 0.25$. (b) Although all modified features are the same between the two cases, they have different data loss rates of the chain-2 service.

The key differences caused by larger placement problems are the out-of-distribution performance measures. For example, the end-to-end latency of a service chain could be greater than any seen in training samples because of its longer execution sequence. Moreover, the system throughput of this service chain could be lower than any seen due to consecutive losses along the longer execution sequence.

To adapt to out-of-distribution performance measures, we make two changes: (i) Instead of learning system throughput X_i and end-to-end latency L_i directly, we learn the ratio between system throughput and arrival rate, and the ratio between total processing time and end-to-end latency. The two ratios are strictly between 0 and 1 because of the occurrence of loss and queueing time, respectively; (ii) Because the latent representation of the end-to-end latency of a service chain is the sum of its fragment node embeddings, a direct sum of an unseen larger number of fragments can cause an out-of-distribution embedding. To overcome this problem, we change the sum into the average of embeddings $\mathbf{h}_{L_i} = \frac{1}{T_i} \sum_{j=1}^{T_i} \mathbf{h}_j^{(N)}$.

Out-of-distribution node features

As described in Section 5.4.2, the input node features are λ_i , $\frac{r_{i,j}}{R_k}$, $m_{i,j}$, and M_k . For ease of presentation, we use $t_{p_{i,j}}$ to denote the processing time $\frac{r_{i,j}}{R_k}$. Our model should have the generalization capability for graph representations where node features come from distributions different from those in training samples. Recall that the service node embedding is recurrently

Table 5.2: Feature modifications for generalization.

	GNN output		$\mathbf{h}_i^{(1),0}$	$\mathbf{h}_j^{(0)}$		$\mathbf{h}_k^{(0)}$
ori	X_i	L_i	λ_i	$t_{p_{i,j}}$	$m_{i,j}$	M_k
md	$\frac{X_i}{\lambda_i}$	$\frac{\sum_{j=1}^{T_i} t_{p_{i,j}}}{L_i}$	1	$\frac{t_{p_{i,j}}}{\lambda_i^{-1}}$	$\frac{t_{p_{i,j}}}{\Delta t_k}$	$\frac{m_{i,j}}{M_k}$
					$\frac{\Delta m_k}{M_k}$	

Legend: ori: original features. md: modified features.

$$\Delta t_k = \sum_{i=1}^C \sum_{j=1}^{T_i} p_{i,j,k} t_{p_{i,j}}$$

updated along the execution sequence, which is the latent representation of the probability of service not being dropped, for the input feature of the chain- i service node, we change λ_i into 1 indicating that no drop happens at the beginning.

To deal with the potential out-of-distribution processing time of a fragment, we change the feature into a ratio between the processing time of this fragment and the interarrival time of its service chain, that is $\frac{t_{p_{i,j}}}{\lambda_i^{-1}}$. This enables us to handle unseen processing times as long as we learn a variety of ratios.

However, this single modification is not enough when a device is shared by multiple service chains. For example, Figure 5.10 shows two cases that have the same modified features, but the data loss rates of the chain-2 service are different. To differentiate such cases, we add one more feature which is the proportion of the fragment processing time $t_{p_{i,j}}$ to the sum of processing times of all types of fragments executed on the shared device, Δt_k . The memory demand $m_{i,j}$ and the maximum memory capacity M_k are changed into ratios: $\frac{m_{i,j}}{M_k}$ and $\frac{\Delta m_k}{M_k}$, respectively. All feature modifications are summarized in Table 5.2.

5.6 Surrogate optimization method

The proposed surrogate optimization algorithm seeks to maximize the constrained program that we have introduced in (5.2). The underpinning search strategy relies on simulated annealing (SA) [NJ10]. SA is a classic search algorithm that, compared to gradient search methods, it is known to escape local minima and find solutions closer to the global optimum.

In our setting, SA starts with an initial placement decision and a temperature parameter τ_0 . At each search step, we consider a new candidate decision that satisfies the constraints of (5.2) and evaluate using the GNN its associated total throughput X_{total} . Thus, the GNN surrogate gives an approximation for the objective function value in (5.2).

At each step, the new candidate decision is generated as follows. We first randomly pick a service chain c and select a DNN fragment for this service running on device k in the current placement. Then, in the new candidate placement we move this fragment to a randomly selected device $k' \neq k$ chosen among those do not already execute fragments of service chain c . For the placement problem to be non-trivial, it is reasonable to assume that the number of available devices exceeds the maximal number of fragments within any service chain, which implies that a device $k' \neq k$ always exists for any fragment. If device k' is already used by $F_{k'}$ fragments of other service chains, we randomly choose b , $0 \leq b \leq F_{k'}$, of those fragments and move them to device k . Thus, the b fragments and the original fragment of service chain c are swapped compared to the original placement.

At step s , the candidate placement $p' \in \mathcal{P}$ is accepted as the new current decision $p \in \mathcal{P}$ if it has a higher total throughput $X_{total} \equiv X_{total}(p')$ than that of p or otherwise SA accepts a worse decision with probability $e^{(X_{total}(p') - X_{total}(p))/\tau_s}$. This scheme is iterated, decreasing the temperature geometrically as $\tau_{s+1} = \gamma\tau_s$, where $\gamma \in (0, 1)$ is a given cooling rate. The algorithm eventually returns the best decision found after exceeding the maximum allowed number of search steps specified by the user. The benefit of its combination with a GNN surrogate is to speed up the search without incurring significant approximation error, so as to increase the likelihood of finding a near-optimal placement.

5.7 Evaluation

In this section, the proposed approach is evaluated in three steps. First, we evaluate the prediction performance of the ChainNet and compare it to baseline GNN models from the literature. Then, we evaluate its generalization ability through an ablation study. Finally, we

Table 5.3: Parameters used for network generation.

Parameter	Type I	Type II
max # devices	10	80
max # service chains	3	12
max # fragments per service chain	6	12
mean interarrival time (λ_i^{-1})	U(0.1,10)	APH(2,5)
fragment processing time ($t_{p_{i,j}}$)	U(0,2)	APH(0.1,10)
maximum memory capacity (M_k)	50	100

The lower bounds for Type II λ_i^{-1} and $t_{p_{i,j}}$ are set to 1 and 0.05.
 λ_i^{-1} and $t_{p_{i,j}}$ are sampled from the listed statistical distributions.

embed the customized GNN model into our surrogate optimization method to demonstrate the benefit of our approach on a large number of placement problems in comparison with the baseline method.

5.7.1 Experiment setup

Training and test data

We obtain a dataset for training and evaluation by simulating 70,000 queueing network models with finite capacity constraints using the Java Modeling Tool (JMT) simulator for QN models [BCS09]. Each JMT simulation abstracts the real deployment in a way similar to Figure 5.2 for the performance of a given placement decision. After discarding the initial transient, the simulator collects 7×10^5 samples of throughputs and latency metrics. Once a simulation is completed, we record the system throughput and end-to-end latency for every service chain in the sample.

The network topologies are randomly generated using two groups of parameters, shown in Table 5.3, and denoted as Type I parameters and Type II parameters. These two sets of parameters represent two classes of edge service systems with different sizes. Type II systems are larger and have more complicated dependencies. Each parameter is chosen uniformly at random, except for the Type II parameters indicated in the table, which are drawn from an acyclic phase-type (APH(μ, s^2)) distribution with mean μ and squared coefficient of variation $s^2 = Var/\mu^2$. The APH distribution has positive support and allows us a fine control of the

Table 5.4: GNN hyperparameters.

Hyperparameter	Value
# iterations/layers	8 (ChainNet, GAT), 12 (GIN)
# hidden layer neurons	64
# attention heads	2
batch size	128
epochs	200
optimizer	Adam [KB14]
learning rate α	0.001, decay 10% per 10 epochs

Table 5.5: Throughput APE results by ChainNet and baseline models.

model	APE (Type I)			APE (Type II)		
	75th	95th	99th	75th	95th	99th
ChainNet	0.012	0.108	0.388	0.011	0.038	0.144
GIN	0.035	0.227	0.688	0.797	0.961	0.987
GAT	0.026	0.219	0.709	0.014	0.112	0.346
GIN*	0.065	0.295	0.945	0.648	1.132	2.210
GAT*	0.040	0.287	0.931	0.083	0.363	1.258

variance of the Type II samples. In the simulations we assume that the execution of a fragment requires a fixed unit of memory, however the number of fragments on a single device can vary allowing to exceed the available memory capacity.

The overall dataset produced in this way corresponds approximately to a week worth of simulations on a cluster of 10 machines and consists of three parts: (i) 50,000 Type I samples for training and validation; (ii) 10,000 Type I samples for test; (iii) 10,000 Type II samples for test. The model is trained using the dataset (i). The samples in test datasets (ii) and (iii) are unseen during the training phase. The purpose of training on Type I samples and testing on Type II samples is to evaluate the OOD generalization ability of the model.

5.7.2 Evaluating the proposed GNN model

Performance metric

For a service chain, the prediction of its performance by the proposed GNN model is evaluated by the *absolute percentage error* (APE), given by $\left| \frac{P-G}{G} \right| \times 100\%$, where P and G are the predicted

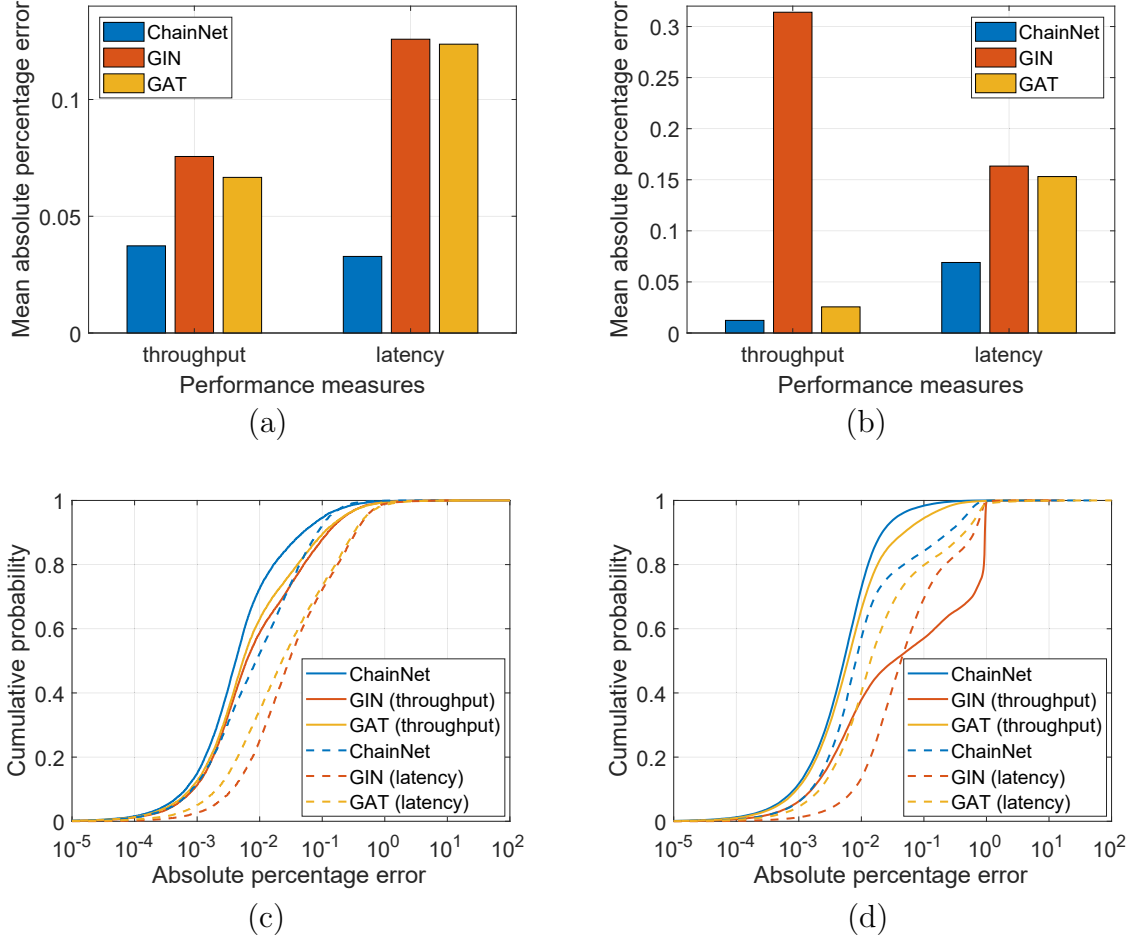


Figure 5.11: (a)-(b): MAPE results on Type I and II test datasets, respectively. (c)-(d): APE distributions on Type I and II test datasets, respectively.

performance measure and ground truth, respectively. Because we predict the performance of every service chain in a collection of network samples, we obtain a set of APEs. We refer in the following to the associated statistical distribution of the APE values as the APE distribution. In addition, we use *mean absolute percentage error* (MAPE) as a metric for the overall prediction performance:

$$\text{MAPE} = \frac{100\%}{Q_{\text{test}}} \sum_{i=1}^{Q_{\text{test}}} \left| \frac{P_i - G_i}{G_i} \right|, \quad (5.17)$$

where Q_{test} is the number of service chains in the test set, i.e., $Q_{\text{test}} = \sum_{g=1}^{G_{\text{test}}} C_g$, and G_{test} is the number of test samples.

Baseline

We compare our customized GNN model against two baseline models: graph isomorphism

network (GIN) and graph attention network (GAT), which are two widely used GNN models adopted for different tasks arising in various application domains of AI [KY20, AAB⁺22, FLY⁺23, HDXD23]. The GIN architecture is designed based on a graph isomorphism test to ensure better discrimination between different graphs. The GAT architecture involves self-attention that allows nodes to attend to distinct neighbors with varying importance. In both architectures, nodes iteratively exchange messages with neighbors. To enable their operations on the constructed graph representation, we connect each service node to its device nodes and fragment nodes using real edges, perform message passing, and make predictions based on service node embeddings. Motivated by DNN chains in edge AI services, ChainNet partitions a graph into multiple execution sequences consisting of execution steps. It performs message-passing in two dimensions: within execution steps and along execution sequences, capturing inter-dependencies between performance metrics within the system. After basic hyperparameter tuning, the configurations adopted by ChainNet and baseline models are shown in Table. 5.4.

Results

Figure 5.11 shows the MAPE and APE distributions for the predicted system throughput and end-to-end latency in Type I and II test datasets. Compared to the best results of the baseline models, ChainNet delivers a substantial error reduction of 48.0% and 64.2% on throughput and latency, respectively. The percentiles of the APE distribution on throughput predictions are shown in Table 5.5. In the table, the value at the i th percentile indicates $i\%$ of the APEs are lower than this value. For instance, it can be observed that 95% of the APEs by ChainNet are below 3.8%. For a fair comparison, the baseline models adopt the same feature modifications proposed in Table 5.2. In Table 5.5, we also provide the results of baseline models when using original features in Table 5.2, which are denoted as GIN* and GAT*. The accuracy of ChainNet is significantly higher than that of all baseline models. In addition to accuracy, a distinct advantage of ChainNet is that it only requires one training phase and then predicts both throughput and latency concurrently, while the other models require separate training phases and predictions.

We then classify the Type II test samples according to the number of nodes and the number of

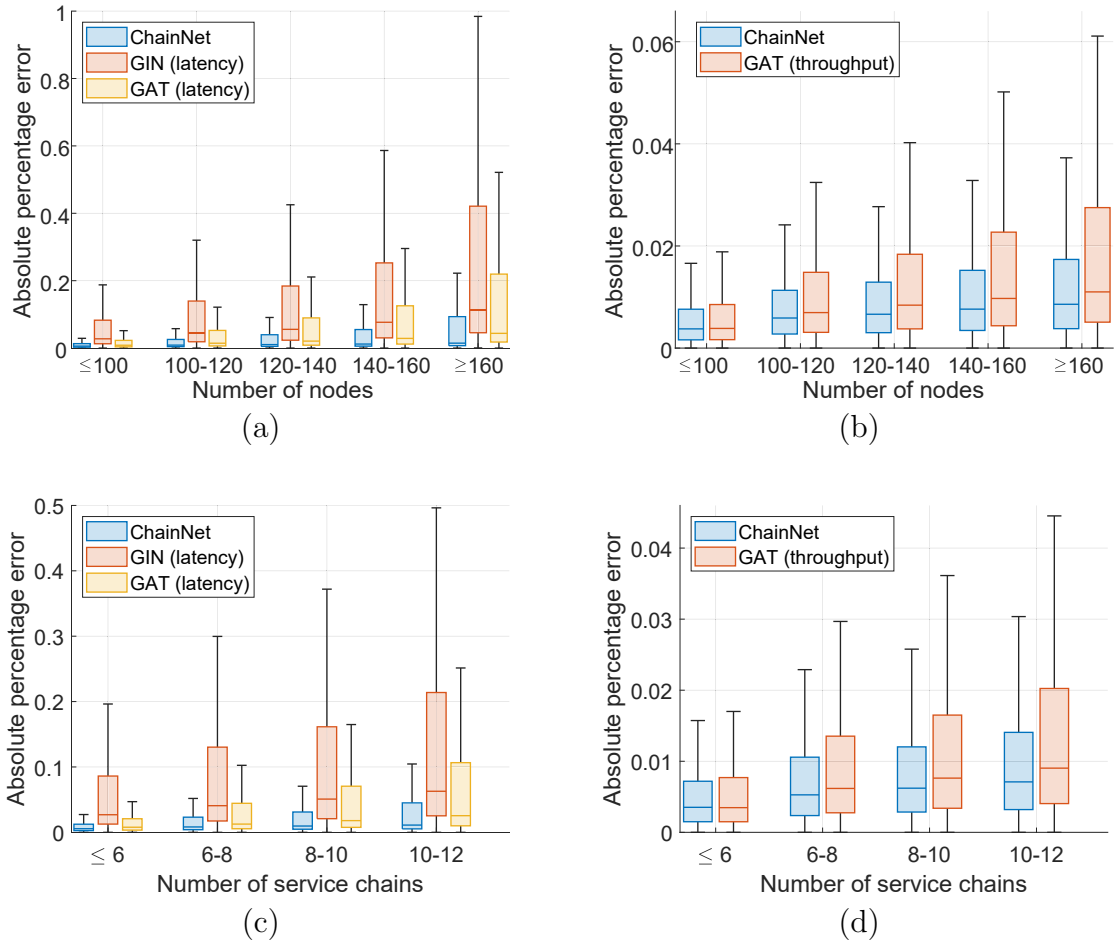


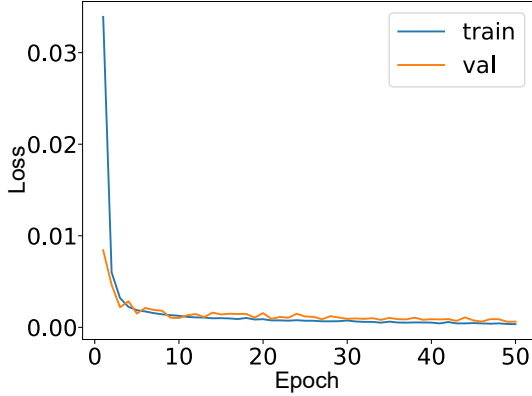
Figure 5.12: Box plot of APE distributions under different number of nodes and service chains. The horizontal line that splits the box is the median. We only present the throughput prediction error boxes of ChainNet and GAT in Figure 5.12b and Figure 5.12d. This is because the medians of GIN boxes are higher than the third quartiles of other boxes, making it inconvenient to display them alongside ChainNet and GAT.

service chains, followed by an evaluation of the prediction performance of GNN models on these groups. Recall that in our graph representation, the number of nodes is the sum of all service chains, fragments, and used devices. The average prediction time per graph is approximately 0.01 seconds across five distinct groups classified by number of nodes, showing no significant difference. As shown in Figure 5.12, the proposed ChainNet consistently demonstrates superior performance compared to the baseline GNN models across all groups. The advantage of our approach becomes more pronounced as the complexity of the test samples increases. These gains demonstrate the effectiveness of the customized design of ChainNet.

Ablation study

Table 5.6: MAPE results by ChainNet and its ablated variants.

model	MAPE (Type I)		MAPE (Type II)	
	throughput	latency	throughput	latency
ChainNet	0.037	0.033	0.012	0.069
ChainNet- α	0.136	0.124	0.213	3.952
ChainNet- β	0.379	0.159	0.794	4.546
ChainNet- δ	0.042	0.050	0.033	0.237



(a) ChainNet

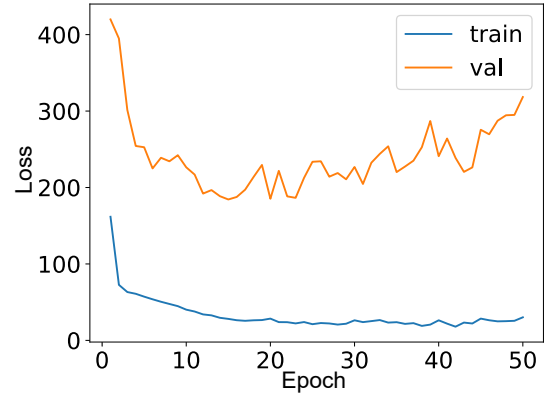
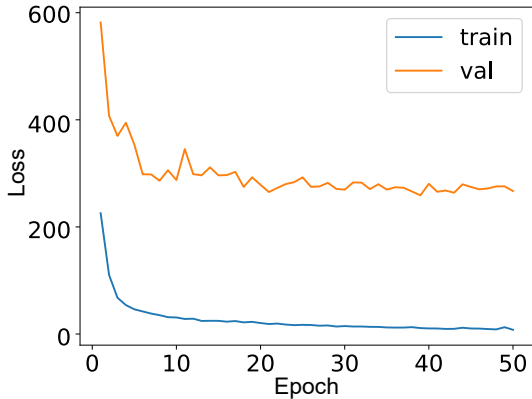
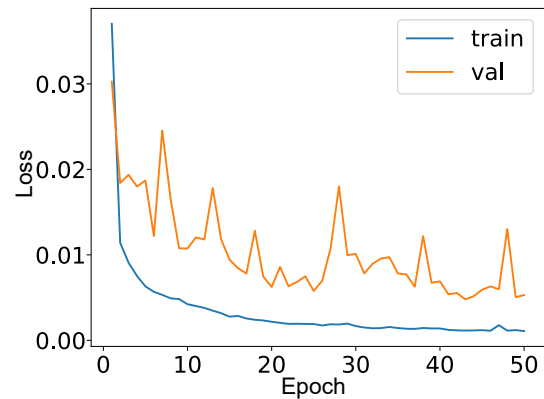
(b) ChainNet- α (c) ChainNet- β (d) ChainNet- δ

Figure 5.13: Ablation study. Variants α , β , and δ do not use the modifications in Table 5.2. A major degradation appears in their validation losses supporting the evidence that removal of GNN features degrades ChainNet.

The generalization method proposed in Table 5.2 consists of two modifications: correcting (i) GNN output and (ii) input initialization, which enables ChainNet to work on large-scale graphs with out-of-distribution node features. In order to assess the effect of each modification, an ablation study is conducted. We compare the prediction error of ChainNet in Type II dataset with that of three ablated versions of the customized GNN: (i) ChainNet- α : without any of the modifications listed in Table 5.2; (ii) ChainNet- β : without the modification of the GNN

Table 5.7: Parameters used for placement problem generation.

Parameter	Value
# available devices	20, 40, 80, 120
# service chains	12
max # fragments per service chain	12
mean interarrival time (λ_i^{-1})	Exp(1)
device service rate (R_k)	U(0.5,1)
maximum memory capacity (M_k)	100
fragment computational demand ($r_{i,j}$)	U(0.01,0.1)

The lower bound of λ_i^{-1} is set to 0.01. λ_i^{-1} , R_k , and $r_{i,j}$ are sampled from the listed distributions.

outputs in Table 5.2; (iii) ChainNet- δ : without the modifications for the input $\mathbf{h}_i^{(1),0}$, $\mathbf{h}_j^{(0)}$, and $\mathbf{h}_k^{(0)}$ in Table 5.2. The MAPE given by ChainNet and its three variants is shown in Table 5.6. As can be observed from ChainNet- β , ablating GNN output modification can cause the largest error increase. Meanwhile, the modified input plays an important role in generalization since ChainNet reduces the MAPE by 63.6% and 70.9% on throughput and latency, respectively, compared to ChainNet- δ .

To visualize the impact of each modification, we use Type II dataset to validate the model during the training process on Type I dataset. Figure 5.13 shows the training and validation loss over the training epochs. The figure emphasizes the critical role of each modification in our generalization design. As can be observed, every element of the design significantly contributes to attaining good generalization ability in large-sized scenarios with distribution shifts. The ablated model variants result in validation losses that either have much larger errors or remain hard to converge.

5.7.3 Evaluating the surrogate optimization program

Placement problems

In this section, we solve 100 randomly generated placement problems, each corresponding to solving an optimization program (5.2), using our surrogate method developed in Section 5.6. Each problem has 12 service chains to be deployed. Every service chain has a variable number

of fragments, up to 12, so that a problem can be formulated at maximum on 144 fragments. The number of available devices in a placement problem is varied as 20, 40, 80, or 120. The parameters used to generate placement problems are summarized in Table 5.7.

Initial placement and search setting

To initialize a placement that triggers the search, we first define the ranking score for the available devices, which follows the rules: (i) The ranking score of an unused device is always greater than that of a used device. (ii) Within each of the two device groups, the ranking score of a device is determined by its remaining memory capacity. Then, we sort the available devices based on their ranking scores from the largest to the smallest. At each assignment iteration, we select a device with the largest ranking score for a fragment, followed by updating its ranking score and re-sorting the devices. We iterate this process until all fragments are assigned. The rationale of this initialization strategy is to use as many devices as it can, which serves as a vanilla deployment that pursues a lower loss rate.

In the optimization process, we carry out a search with a maximum of 100 steps. We refer to each search trajectory as a *trial*. After completing a trial, the search starts again from the same initial placement. Since the SA method in Section 5.6 is randomized, this results in a different search trajectory. We find that a cooling rate $\gamma = 0.9$ gives good performance for both ChainNet and the baseline optimization programs.

Figure 5.14a gives an example of SA run showing the search trajectories for five trials. As we can see, the rate of data loss reduction is highly variable and the best placement decision is ultimately given by the second trial. The curves of trials 1, 3, 4, and 5 are fairly similar, while the trial 2 curve has two sharp decreases around the 30th and 60th steps. While it is well known that multi-start methods that attempt repeated trials improve SA solutions, this example indicates that this remains the case also for the placement problem we consider. As we show later, the key benefit of ChainNet is to allow to run many more trials per unit time than simulation-based approaches, while attaining a very precise surrogate representation of the search surface, allowing high fidelity in the representation of the objective function.

Performance metric

We define the loss probability of a placement p as

$$\pi_{loss}(p) = \frac{\lambda_{total} - X_{total}(p)}{\lambda_{total}}, \quad (5.18)$$

where $\lambda_{total} = \sum_{i=1}^C \lambda_i$, and $X_{total}(p)$ is the objective function value at p given in (5.2).

Since the goal of our optimization program is to minimize the overall loss rate, we use the relative loss reduction to evaluate the optimized placement p , which is defined as

$$\eta(p) = \frac{\pi_{loss}(p_0) - \pi_{loss}(p)}{\pi_{loss}(p_0)} = \frac{X_{total}(p) - X_{total}(p_0)}{\lambda_{total} - X_{total}(p_0)}, \quad (5.19)$$

where p_0 is the initial placement from which the search starts.

We consider the mean loss probability and mean relative loss reduction achieved by the optimization program on generated placement problems.

Baseline

We now compare on the 100 random placement problems the performance of the ChainNet-based optimization program with that of the JMT simulation-based optimization program. We split the validation into two experiments that are conducted separately:

Fixed-time optimization We compare the mean relative loss reduction achieved by ChainNet-based search to that achieved by simulation-based search after a fixed time length, corresponding to the duration of one simulation-based trial. Since GNN-based estimation of latency and throughput is faster than simulation, the ChainNet-based method is able to run multiple trials in the period during which the simulation completes its single trial and we compare the best solution found by both methods in the same time period.

Fixed-steps optimization We compare the mean relative loss reduction and optimization duration of ChainNet-based search to that of simulation-based search fixing the number of

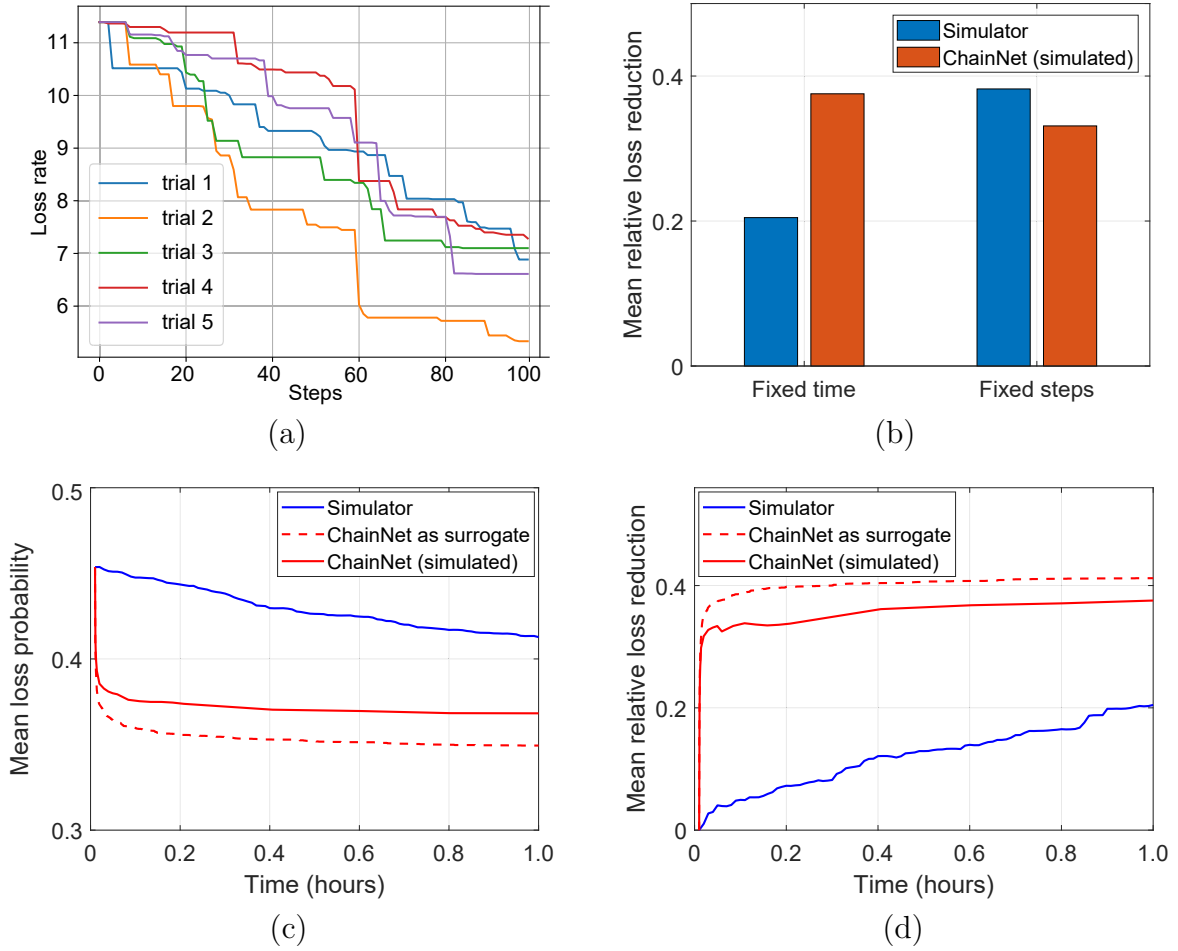


Figure 5.14: (a): An example of different objective function reductions across 5 independent trials. (b): Mean relative loss reduction by ChainNet and baseline method in two groups of experiments. (c)-(d): The change process of mean loss probability and mean relative loss reduction over the fixed time frame. Dashed lines represent estimated results by ChainNet, while solid red lines represent simulated results of ChainNet decisions. The curves show a steep improvement of ChainNet in a short amount of time.

search steps to 30 trials for both GNN and simulator, thus comprising total 3,000 steps. As the GNN is a surrogate of the simulated data, it is expected that an accurate GNN model should display very similar result as the simulator in this test. Thus we ask whether this is the case or whether the GNN faces some distortion of the search surface that misleads the optimization.

Results

Given a placement problem, the ChainNet-based optimization program provides us with an optimized decision. After completion of the GNN-based optimization, we post-process the results to correctly characterize its performance as follows: the GNN solution is passed to

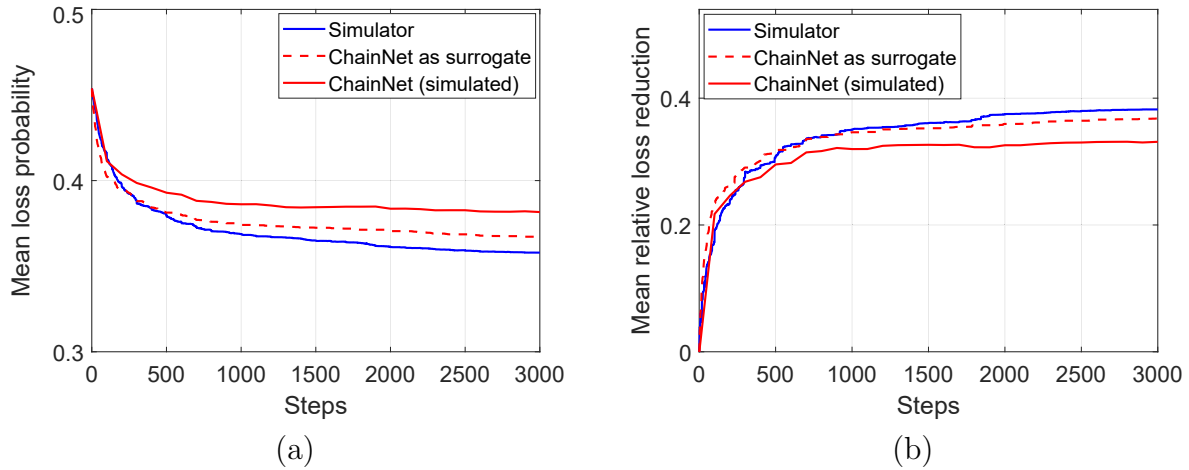


Figure 5.15: (a)-(b): The change process of mean loss probability and mean relative loss reduction over the same number of steps. The curves indicate that ChainNet is accurate in capturing trends of simulation-based search.

the simulator, then the actual performance metrics of this decision are collected and used to produce the results reported in this section. Thus, we do not report the loss rates estimated by the GNN itself, which we observe are often optimistic, though close to the simulated values, and conservatively use instead the simulated values obtained in post-processing.

Overall results Figure 5.14b shows the mean relative loss reduction gained by ChainNet and simulation-based baseline method. Recall that for each problem, both searches start from the same initial placement. For the results of fixed-time optimization, the baseline and ChainNet achieve 20.5% and 37.6%, respectively. ChainNet delivers a 83.4% improvement compared to the simulation-based method. When conducting the optimization using the same number of steps, ChainNet achieves 86.7% of the level of the baseline method, with a high time efficiency.

Fixed-time optimization We give more details on this scenario by plotting the mean loss probability and mean loss relative reduction along the fixed time frame (Figure 5.14c to 5.14d). As can be observed, in the fixed-time optimization, the ChainNet-based program shows a steep improvement in the curve in the very early iterations. The mean loss reduction by ChainNet exceeds 35% before 0.1 hours (6 minutes), while the simulation-based search provides a slight 5% loss reduction at this time point. The overall curve of the objective function given by ChainNet dominated the simulation-based curve.

Fixed-steps optimization For this scenario, we plot the change of the two metrics over the same number of steps (Figure 5.15a to 5.15b). It can be observed that the evolution of loss rate in simulation-based search is accurately captured by ChainNet. Moreover, in such cases, the ChainNet-based program is far more efficient than the baseline. The simulation-based program takes around 30 hours to decide the best optimal placement, while the ChainNet-based program only takes 90 seconds to complete an optimization of 30 trials. Another observation is that the tails of both curves get slow to improve; this is due to the relative difficulty of improving the optimum through randomization as the search continues.

5.7.4 A case study with real-world parameters

The proposed ChainNet optimization program is now applied to the simulated placement of DNNs based on parameters from a real-world technological scenario comprising $2 \times$ OrangePi Zero, $2 \times$ Raspberry Pi A+, and $1 \times$ Raspberry Pi 3A+, with 128 MB, 256 MB, and 512 MB of RAM, and 4.8 GFLOPs, 0.218 GFLOPs, 5 GFLOPs of service rates, respectively. We seek to optimally deploy 8 deep neural networks of 4 types: a VGG16, a VGG19, a 28-layer custom CNN for image classification, and a custom CNN for intrusion detection from [CR23]. Each model is partitioned into three and four fragments, forming two distinct service chains. Thus the problem has a total of $2 \times 4 = 8$ chains that require the deployment of 28 fragments. The memory and computational demands of these fragments range from 4 KB to 51879 KB, and 0.6 FLOPs to 396.8 FLOPs. FLOPs are converted into edge device processing times based on their nominal speeds (in GHz). The exponential inter-arrival times of 3-fragment and 4-fragment networks have mean of 0.6s and 0.7s, respectively. We conduct a simulation-based study with JMT and estimate that the initial deployment experiences a loss probability of 96.2%. We then perform a 100-step ChainNet-based optimization (taking around 3s) that reduces the loss probability to 14.6%, while the simulation-based optimization reduces it only to 86.8% in a much longer time span of 10 minutes. In addition, a 100-step GAT and GIN-based optimization achieves a loss probability of 23.5% and 94.7%, respectively. These results reinforce the OOD generalization ability of ChainNet.

5.8 Summary

In this chapter, we propose ChainNet, a novel customized GNN that can be used as a surrogate for loss-aware edge AI deployment, which shows promise in evaluating and settling complex chain-based services. We summarize the main points as follows:

- ChainNet models a distributed edge AI system using a fine-grained graph that reflects services, fragments, and devices, as well as their placement and workflow relationships. The customization of its message passing is informed by queueing analysis, specially tailored to open multiclass QNs where resource congestion and job loss occur.
- We analyze how performance changes as the system increases in complexity and present a comprehensive design that equips ChainNet with OOD generalization capabilities for larger graphs and node features subject to distribution shift.
- We compare the prediction performance of ChainNet with that of GIN and GAT. ChainNet consistently demonstrates more accurate predictions, achieving an average improvement of over 50%, which highlights the effectiveness of our customized design.
- We conduct an ablation study to assess the effect of each component of our OOD design. The results indicate a significant decrease in prediction accuracy when any component is removed, confirming that each plays a critical role in dealing with distribution shifts.
- We assess ChainNet-based and simulation-based optimization programs under fixed-time and fixed-step conditions. ChainNet enhances the simulation outcomes by over 83% within a fixed timeframe and efficiently captures the loss changes across the same steps.
- We implement loss-aware optimizations based on different surrogates for a placement task with real-world technological parameters. The minimum loss attained by ChainNet-based program is nearly 40% lower than that of the next best surrogate.

Chapter 6

Conclusion

In this closing part, we summarize the achievements of each technical chapter in this thesis and outline potential directions for future work based on the current contributions.

6.1 Summary of Thesis Achievements

In Chapter 3, we address the problem of estimating response time densities in layered queueing networks (LQNs) by proposing an innovative approach that integrates mixture density networks (MDNs) with a phase-type (PH) analytical framework. First, we employ the stochastic rendezvous network (SRVN) layering strategy to decompose an LQN model into a set of layers, each represented as a machine repairman queueing network. Then, we predict and propagate response time densities from the bottom layer up to the top layer, ultimately obtaining the end-to-end system response time density. Within each layer, we first characterize the service demand of an arbitrary activity graph based on PH distributions. We then use an MDN to predict the response time distribution, fit a PH distribution through moment matching, and pass it to the upper layer that sends the service requests. We propose using PH distributions to model the durations of service requests for passing densities between layers, resulting in an end-to-end PH framework that ensures the tractability of probability distributions throughout the entire layered system. Additionally, we propose a PH modeling approach for the residence

time within a loop, allowing us to handle more diverse activity graphs. In experiments, we first demonstrate the high accuracy of MDNs in predicting response time distributions in queueing networks, showing that the MDN is a strong candidate for tackling density estimation problems in queueing theory. We then integrate trained MDN analyzers into our PH density-passing framework and evaluate its performance using LQNs with complex activity graphs and interconnections. The high accuracy of the predictions validates the effectiveness of our combined use of the following techniques: MDN predictions, PH service demand characterization, and PH service request modeling. Finally, we conduct layering depth experiments to assess the approach across a range of LQNs with varying topologies and increasing depths. The results highlight that our method significantly reduces prediction errors compared to state-of-the-art analytical methods, with the performance gap widening as more layers are added.

In Chapter 4, we use graph neural networks (GNNs) to reveal the potential of learning-based models as a general-purpose solver for different queueing networks. We propose a graph learning-based solver capable of predicting main performance measures (i.e., throughput, response time, queue length, utilization) per job class at each service center in a closed multiclass queueing network. To the best of our knowledge, this is the first attempt to use a learning-based end-to-end solver for predicting a broad range of queueing networks with a flexible number of job classes, number of service centers, and different topologies. To evaluate the performance of a queueing network, we first represent it as a graph that serves as the input to our GNN solver. The graph construction reflects two key components of the queueing network: job class and service center, represented by two types of nodes called global nodes and local nodes. Our solver predicts the system performance of each job class from its global node embedding and the center performance of each job class from the local node embedding. To obtain these embeddings, we analyze the interdependencies inside queueing networks and develop message-passing processes conducted on two subgraphs: global graph and local graph. In the global graph, we update each global node embedding by learning the distinct impacts of service centers on a job class. In the local graph, we update each local node embedding by differentiating the effects of job classes on the shared service center. In experiments, we first focus on two-class machine repairman queueing networks that ordinary MLPs can effectively solve. We demonstrate that

both MLPs and GNNs can make accurate predictions. We then extend our study to networks with varying numbers of job classes, where GNNs continue to demonstrate the expected accuracy. Subsequently, we propose GNN variants to enhance this accuracy, providing insights that learning both system and center performance concurrently can lead to better results. We then tackle general closed multiclass queueing networks and validate the effectiveness of our solver. Finally, we compare the performance of the solver with that of a standard off-the-shelf GNN, the results suggest the advantages of our approach for queueing problems.

In Chapter 5, we first abstract open multiclass queueing networks, which experience losses due to finite shared queues, from the edge AI service. Next, we recognize three elements (i.e., service, fragment, and device) along with two relations (i.e., placement and workflow), which together form a corresponding graph. Drawing inspiration from DNN chains in edge AI services, we partition this graph into multiple execution sequences, each composed of specific execution steps. We then analyze the execution behavior and interdependencies among these steps, establishing latent representations for both throughput and latency. Meanwhile, we examine the interdependencies within the execution step and customize the message-passing process. Moreover, we propose a detailed design that enables ChainNet to achieve out-of-distribution (OOD) generalization on larger placement tasks and with node features drawn from different distributions. Finally, we integrate a trained ChainNet, as a surrogate model, into the simulated annealing algorithm to search for the deployment that minimizes data loss. In experiments, we first show that ChainNet performs significantly better than established GNN models, suggesting the effectiveness of our customized design. Following that, we conduct an ablation study that confirms the contribution of individual design components to the overall performance, highlighting the OOD generalization capability. We then evaluate the performance of the ChainNet-based optimization program against the simulation-based optimization program under both fixed-time and fixed-steps scenarios. For fixed-time optimization, ChainNet enables considerably improved decisions within a set timeframe. For fixed-steps optimization, ChainNet accurately tracks the evolution of the loss rate observed in simulation-based searches and demonstrates much greater efficiency. Finally, we apply the ChainNet-based optimization program for DNN placement with real-world technological parameters, demonstrating its ad-

vantages over established GNNs and the simulation-based optimization program and reinforcing its OOD generalization performance.

6.2 Future Work

Building on the work completed in Chapter 3, two main future directions may be explored. The first direction involves using machine learning approaches. Given the demonstrated advantages of GNN approaches in Chapter 4 and Chapter 5, we could apply a GNN to encode machine repairman models or even the entire LQN and then combine it with a mixture distribution to directly learn the response time distribution. The second direction focuses on incorporating additional features of LQNs, which can be explored through the following aspects: (i) considering *second phase* [LNW88] of service, an LQN extension developed to reduce the blocking time of clients during RPC interactions. It may be possible to adapt our method to this case by training MDNs on simulations that include the second phase feature, (ii) including fork/join nodes in activity graphs, as they are sensitive to response time distribution, (iii) adding job impatience so that calls are rerouted or dropped after a timeout threshold, (iv) incorporating forwarding and asynchronous calls (i.e., open job classes), (v) studying the effect of interlocking [Fra00] on our approach, a phenomenon where calls may jump over layers.

Next, based on the research conducted in Chapter 4, we propose three possible future directions. The first direction involves expanding our GNN solver to support class-switching processes, where jobs can switch between different classes and change their behaviors and service demands. This direction poses two open questions that may be challenging to resolve: (i) how to represent a class-switching process in a graph, and (ii) what message-passing design would enable the GNN to more effectively distinguish the unique impact of each job class when class boundaries are blurred due to class-switching. A potential solution could be combining a GNN with queueing analysis techniques that aggregate communicating job classes into a chain. The second direction is to incorporate priority networks, where certain job classes have priority over others, and within each priority class, the scheduling discipline follows FCFS. This can be implemented

by encoding priority into the global node embedding of each job class, though various encoding strategies should be explored. The third direction aims to extend our GNN solver to deal with queueing networks whose service centers have a mixture of scheduling disciplines, an initial exploration shows promising results in Chapter 4.4.5.

Lastly, to continue the work presented in Chapter 5, we suggest two important potential directions. The first direction is an extension of ChainNet. The DNN models considered in Chapter 5 are constructed as a chain of sequentially executed fragments, forming a deterministic routing structure that shapes the design of ChainNet. However, in less frequent scenarios, DNN models in Edge AI may deviate from strict sequential execution, such as networks with non-linear structures [HL22, DXJ⁺23] or early-exit mechanisms for adaptive inference [BWDS17]. To accommodate these scenarios, an extension of ChainNet may incorporate probabilistic and parallel routing. The second direction involves enriching ChainNet-based optimization programs to benefit other tasks, such as real-time resource management. This could include integrating ChainNet with online optimization techniques [MSV20, Pad21], which makes it possible for ChainNet to help reroute traffic in edge AI service networks to recover from unsuccessful transmissions caused by link failures [AB20].

Bibliography

- [AAB⁺22] Teresa Alsinet, Josep Argelich, Ramón Béjar, Daniel Gibert, and Jordi Planes. Argumentation reasoning with graph isomorphism networks for reddit conversation analysis. *International Journal of Computational Intelligence Systems*, 15(1):86, 2022.
- [AB20] Atakan Aral and Ivona Brandić. Learning spatiotemporal failure dependencies for resilient edge computing services. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1578–1590, 2020.
- [Alp21] Ethem Alpaydin. *Machine learning*. MIT press, 2021.
- [Bar79] Yonathan Bard. Some extensions to multiclass queueing network analysis. In *Proceedings of the Third International Symposium on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems*, pages 51–62, 1979.
- [BAY21] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2021.
- [BB80] Steven C Bruell and Gianfranco Balbo. *Computational algorithms for closed queueing networks*. Elsevier North-Holland, 1980.
- [BB23] Christopher M Bishop and Hugh Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023.

- [BCMP75] Forest Baskett, K Mani Chandy, Richard R Muntz, and Fernando G Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM (JACM)*, 22(2):248–260, 1975.
- [BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.
- [BDJ20] Mounir Bensalem, Jasenka Dizdarević, and Admela Jukan. Modeling of deep neural network (dnn) placement and inference in edge computing. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2020.
- [BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
- [BG17] Axel Brando Guillaumes. Mixture density networks for distribution and uncertainty estimation. Master’s thesis, Universitat Politècnica de Catalunya, 2017.
- [BGDMT06] Gunter Bolch, Stefan Greiner, Hermann De Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.
- [BHT05] Andrea Bobbio, András Horváth, and Miklós Telek. Matching three moments with minimal acyclic phase type distributions. *Stochastic Models*, 21(2-3):303–326, 2005.
- [Bis94] Christopher M. Bishop. Mixture density networks. Workingpaper, Aston University, 1994.
- [BKSS22] Opher Baron, Dmitry Krass, Eliran Sherzer, and Arik Senderovich. Can machines solve general queueing problems? In *2022 Winter Simulation Conference (WSC)*, pages 2830–2841. IEEE, 2022.

- [BPI03] Simonetta Balsamo, Vittoria De Nitto Personè, and Paola Inverardi. A review on queueing network models with finite capacity queues for software architectures performance prediction. *Performance Evaluation*, 51(2-4):269–288, 2003.
- [BS17] Hendrik Baumann and Werner Sandmann. Multi-server tandem queue with markovian arrival process, phase-type service times, and finite buffers. *European Journal of Operational Research*, 256(1):187–195, 2017.
- [BWDS17] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.
- [Cas20] Giuliano Casale. Integrated performance evaluation of extended queueing network models with line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, 2020.
- [CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.
- [CDMI11] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*, volume 980. Springer, 2011.
- [CGNZ22] Giuliano Casale, Yicheng Gao, Zifeng Niu, and Lulai Zhu. Ln: A meta-solver for layered queueing network analysis. In *International Conference on Quantitative Evaluation of Systems*, pages 232–254. Springer, 2022.
- [CGNZ23] Giuliano Casale, Yicheng Gao, Zifeng Niu, and Lulai Zhu. Ln: A flexible algorithmic framework for layered queueing network analysis. *ACM Transactions on Modeling and Computer Simulation*, 2023.
- [CH23] Wang Cen and Peter J Haas. Nim: Generative neural networks for automated modeling and generation of simulation inputs. *ACM Transactions on Modeling and Computer Simulation*, 33(3):1–26, 2023.

- [Cha72] KM Chandy. The analysis and solutions for general queueing networks. In *Proceedings of the Sixth Annual Princeton Conference on Information Sciences and Systems*, pages 224–228, 1972.
- [CHJT77] K Mani Chandy, John H Howard Jr, and Donald F Towsley. Product form and local balance in queueing networks. *Journal of the ACM (JACM)*, 24(2):250–263, 1977.
- [CHW75] K. Mani Chandy, Ulrich Herzog, and L Woo. Approximate analysis of general queueing networks. *IBM Journal of Research and Development*, 19(1):43–49, 1975.
- [CN82] K Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, 1982.
- [CPR19] Florin Ciucu, Felix Poloczek, and Amr Rizk. Queue and loss distributions in finite-buffer queues. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–29, 2019.
- [CR19] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.
- [CR23] Giuliano Casale and Manuel Roveri. Scheduling inputs in early exit neural networks. *IEEE Transactions on Computers*, 2023.
- [CS78] K Mani Chandy and Charles H Sauer. Approximate methods for analyzing queueing network models of computing systems. *ACM Computing Surveys (CSUR)*, 10(3):281–317, 1978.
- [CSK21] George Constantinou, Cyrus Shahabi, and Seon Ho Kim. Placement of dnn models on mobile edge devices for effective video analysis. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 207–218. IEEE, 2021.

- [CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.
- [Cum82] Aldo Cumani. On the canonical representation of homogeneous markov processes modelling failure-time distributions. *Microelectronics Reliability*, 22(3):583–602, 1982.
- [CVMG⁺14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [CYB⁺21] Eunho Cho, Juyeon Yoon, Daehyeon Baek, Dongman Lee, and Doo-Hwan Bae. Dnn model deployment on distributed edges. In *International Conference on Web Engineering*, pages 15–26. Springer, 2021.
- [CZS10] Giuliano Casale, Eddy Z Zhang, and Evgenia Smirni. Kpc-toolbox: Best recipes for automatic trace fitting using markovian arrival processes. *Performance Evaluation*, 67(9):873–896, 2010.
- [DB78] Peter J Denning and Jeffrey P Buzen. The operational analysis of queueing network models. *ACM Computing Surveys (CSUR)*, 10(3):225–261, 1978.
- [DHCI20] Christopher N Davis, T Deirdre Hollingsworth, Quentin Caudron, and Michael A Irvine. The use of mixture density networks in the emulation of complex epidemiological individual-based models. *PLoS Computational Biology*, 16(3):e1006869, 2020.
- [DRA21] Simone Disabato, Manuel Roveri, and Cesare Alippi. Distributed deep convolutional neural networks for the internet-of-things. *IEEE Transactions on Computers*, 70(8):1239–1252, 2021.
- [DXJ⁺23] Xingxia Dai, Zhu Xiao, Hongbo Jiang, Ming Lei, Geyong Min, Jiangchuan Liu, and Schahram Dustdar. Offloading dependent tasks in edge computing with

- unknown system-side information. *IEEE Transactions on Services Computing*, 2023.
- [DXT⁺20] Shuiguang Deng, Zhengzhe Xiang, Javid Taheri, Mohammad Ali Khoshkholghi, Jianwei Yin, Albert Y Zomaya, and Schahram Dustdar. Optimal application deployment in resource constrained distributed edges. *IEEE Transactions on Mobile Computing*, 20(5):1907–1923, 2020.
- [FAOW⁺08] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2008.
- [FC20] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.
- [FGPSV⁺23] Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Krzysztof Rusek, Shihan Xiao, Xiang Shi, Xiangle Cheng, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet-fermi: Network modeling with graph neural networks. *IEEE/ACM Transactions on Networking*, 2023.
- [FLY⁺23] Kimon Fountoulakis, Amit Levi, Shenghao Yang, Aseem Baranwal, and Aukosh Jagannath. Graph attention retrospective. *Journal of Machine Learning Research*, 24(246):1–52, 2023.
- [FML⁺19] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.
- [FMW⁺05] Greg Franks, Peter Maly, Murray Woodside, Dorina C Petriu, Alex Hubbard, and Martin Mroz. Layered queueing network solver and simulator user manual. *Dept. of Systems and Computer Engineering, Carleton University (December 2005)*, pages 15–69, 2005.
- [Fra00] Greg Franks. *Performance analysis of distributed server systems*. PhD thesis, Carleton University, 2000.

- [FWR06] Greg Franks, Murray Woodside, and Jerry Rolia. Multi-threaded servers with high service time variation for layered queueing networks. In *Computer System Performance Modeling In Perspective: A Tribute to the Work of Prof Kenneth C Sevcik*, pages 137–154. World Scientific, 2006.
- [GATW11] Michael Grottke, Varsha Apte, Kishor S Trivedi, and Steve Woolet. Response time distributions in networks of queues. *Queueing Networks: A Fundamental Approach*, pages 587–641, 2011.
- [GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [GCW19] Alim Ul Gias, Giuliano Casale, and Murray Woodside. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019.
- [GIT20] Giulio Garbi, Emilio Incerto, and Mirco Tribastone. Learning queueing networks by recurrent neural networks. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 56–66, 2020.
- [GLME⁺15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges, 2015.
- [GN67] William J Gordon and Gordon F Newell. Closed queueing systems with exponential servers. *Operations research*, 15(2):254–265, 1967.
- [GSR⁺17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.
- [Ham20] William L Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

- [Har93] Peter G Harrison. Response time distributions in queueing network models. In *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 147–164. Springer, 1993.
- [HB13] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.
- [HDXD23] Haiwei Hou, Shifei Ding, Xiao Xu, and Ling Ding. A novel clustering algorithm based on multi-layer features and graph attention networks. *Soft Computing*, 27(9):5553–5566, 2023.
- [He14] Qi-Ming He. *Fundamentals of matrix-analytic methods*, volume 365. Springer, 2014.
- [HK02] Peter G Harrison and William J Knottenbelt. Passage time distributions in large markov chains. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 77–85, 2002.
- [HL22] Chenghao Hu and Baochun Li. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 330–339. IEEE, 2022.
- [HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- [HWP⁺23] Sijiang Huang, Yunze Wei, Lingfeng Peng, Mowei Wang, Linbo Hui, Peng Liu, Zongpeng Du, Zhenhua Liu, and Yong Cui. xnet: Modeling network performance with graph neural networks. *IEEE/ACM Transactions on Networking*, 2023.
- [HYL17] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. 2017.
- [Isl18] Farhana Islam. *Simplifying layered queueing network models*. PhD thesis, Carleton University, 2018.

- [Ism23] Vugar E Ismailov. A three layer neural network can represent any multivariate function. *Journal of Mathematical Analysis and Applications*, 523(1):127096, 2023.
- [Jac63] James R Jackson. Jobshop-like queueing systems. *Management science*, 10(1):131–142, 1963.
- [JC61] Little JD C. A proof of the queueing formula: $L = \lambda w$. *Operations Research*, 9(3):383–387, 1961.
- [JM15] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.
- [KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [KGK⁺22] Martin Kaiser, René Griessl, Nils Kucza, Carola Haumann, Lennart Tigges, Kevin Mika, Jens Hagemeyer, Florian Porrmann, Ulrich Rückert, Micha vor dem Berge, et al. Vedliot: very efficient deep learning in iot. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 963–968. IEEE, 2022.
- [KM02] Hisashi Kobayashi and Brian L Mark. Generalized loss models and queueing-loss networks. *International Transactions in Operational Research*, 9(1):97–112, 2002.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.
- [KT19] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.
- [Kur70] Thomas G Kurtz. Solutions of ordinary differential equations as limits of pure jump markov processes. *Journal of Applied Probability*, 7(1):49–58, 1970.

- [KY20] Byung-Hoon Kim and Jong Chul Ye. Understanding graph isomorphism network for rs-fmri functional connectivity analysis. *Frontiers in Neuroscience*, 14:630, 2020.
- [LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [LHAES23] Qianlin Liang, Walid A Hanafy, Ahmed Ali-Eldin, and Prashant Shenoy. Model-driven cluster resource management for ai workloads in edge clouds. *ACM Transactions on Autonomous and Adaptive Systems*, 18(1):1–26, 2023.
- [LL19] Chen Li and Gim Hee Lee. Generating multiple hypotheses for 3d human pose estimation with mixture density network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9887–9895, 2019.
- [LLL⁺21] Jing Li, Weifa Liang, Yuchen Li, Zichuan Xu, Xiaohua Jia, and Song Guo. Throughput maximization of delay-aware dnn inference in edge computing by exploring dnn model partitioning and inference parallelism. *IEEE Transactions on Mobile Computing*, 2021.
- [LLY⁺21] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 33(12):6999–7019, 2021.
- [LNW88] Louis-Marie Le Ny and C Murray Woodside. *Performance modelling of queues with rendezvous service*. PhD thesis, INRIA, 1988.
- [LR80] Steve S Lavenberg and Martin Reiser. Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers. *Journal of Applied Probability*, 17(4):1048–1061, 1980.
- [LZGS84] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

- [LZZC19] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.
- [MBP⁺23] Lorenzo Manca, Davide Borsatti, Filippo Poltronieri, Mattia Zaccarini, Domenico Scotece, Gianluca Davoli, Luca Foschini, Genady Ya Grabarnik, Larisa Shwartz, Cesare Stefanelli, et al. Characterization of microservice response time in kubernetes: A mixture density network approach. In *2023 19th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2023.
- [MDG21] Seyed Samie Mostafavi, György Dán, and James Gross. Data-driven end-to-end delay violation probability prediction with extreme value mixture models. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 416–422. IEEE, 2021.
- [MJ51] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.
- [MSV20] Sajad Modaresi, Denis Sauré, and Juan Pablo Vielma. Learning in combinatorial optimization: What and how to explore. *Operations Research*, 68(5):1585–1604, 2020.
- [NC81] Doug Neuse and K Chandy. Scat: A heuristic algorithm for queueing network models of computing systems. In *Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 59–79, 1981.
- [NC21] Zifeng Niu and Giuliano Casale. A mixture density network approach to predicting response times in layered systems. In *2021 29th International Symposium On Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2021.
- [NC24] Zifeng Niu and Giuliano Casale. Neural density estimation of response times in layered software systems. *IEEE Transactions on Software Engineering*, 2024.

- [Neu94] Marcel F Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Courier Corporation, North Chelmsford, MA, USA, 1994.
- [Niu24] Zifeng Niu. Graph learning based performance analysis for queueing networks. *ACM SIGMETRICS Performance Evaluation Review*, 51(3):6–7, 2024.
- [NJ10] Alexander G Nikolaev and Sheldon H Jacobson. Simulated annealing. *Handbook of Metaheuristics*, pages 1–39, 2010.
- [NOW20] Suguru Nii, Takashi Okudal, and Takuya Wakita. A performance evaluation of queueing systems by machine learning. In *2020 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-Taiwan)*, pages 1–2. IEEE, 2020.
- [NPD21] Hung Quoc Nguyen and Tuan Phung-Duc. Mixture density networks as a general framework for estimation and prediction of waiting time distributions in queueing systems. In *European Workshop on Performance Engineering*, pages 148–161. Springer, 2021.
- [NRC24] Zifeng Niu, Manuel Roveri, and Giuliano Casale. Chainnet: A customized graph neural network model for loss-aware edge ai service deployment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024.
- [OCG⁺21] César Ojeda, Kostadin Cvejovski, Bodgan Georgiev, Christian Bauckhage, Jannis Schuecker, and Ramsés J Sánchez. Learning deep generative models for queueing systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9214–9222, 2021.
- [ODF07] Tariq Omari, Salem Derisavi, and Greg Franks. Deriving distribution of thread service time in layered queueing networks. In *Proceedings of the 6th International Workshop on Software and Performance*, pages 66–77, 2007.
- [Pad21] Sindhu Padakandla. A survey of reinforcement learning algorithms for dynamically varying environments. *ACM Computing Surveys (CSUR)*, 54(6):1–25, 2021.

- [Pet21] Dorina C Petriu. Integrating the analysis of multiple non-functional properties in model-driven engineering. *Software and Systems Modeling*, 20(6):1777–1791, 2021.
- [PG22] Gopika Premasankar and Bissan Ghaddar. Energy-efficient service placement for latency-sensitive applications in edge computing. *IEEE Internet of Things Journal*, 9(18):17926–17937, 2022.
- [PK22] Arjun Parthasarathy and Bhaskar Krishnamachari. Partitioning and placement of deep neural networks on distributed edge devices to maximize inference throughput. In *2022 32nd International Telecommunication Networks and Applications Conference (ITNAC)*, pages 239–246. IEEE, 2022.
- [PSZ21] Jamol Pender, Sukriti Sudhakar, and Eva Zhang. Combining simulation and machine learning for response time prediction for the shortest remaining processing time discipline. In *2021 Winter Simulation Conference (WSC)*, pages 1–12. IEEE, 2021.
- [RBAC22] Johan Ruuskanen, Tommi Berner, Karl-Erik Arzen, and Anton Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 49(3):69–70, 2022.
- [RBM23] T Konstantin Rusch, Michael M Bronstein, and Siddhartha Mishra. A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv:2303.10993*, 2023.
- [Rei79] Martin Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Transactions on Communications*, 27(8):1199–1209, 1979.
- [RH10] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

- [RK75] Martin Reiser and Hisashi Kobayashi. Queuing networks with multiple closed chains: theory and computational algorithms. *IBM Journal of Research and Development*, 19(3):283–294, 1975.
- [RL80] Martin Reiser and Stephen S Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM (JACM)*, 27(2):313–322, 1980.
- [RRXB23] Manon Réau, Nicolas Renaud, Li C Xue, and Alexandre MJJ Bonvin. Deeprank-gnn: a graph neural network framework to learn patterns in protein-protein interfaces. *Bioinformatics*, 39(1):btac759, 2023.
- [RS95] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995.
- [RTL19] Majid Raeis, Ali Tizghadam, and Alberto Leon-Garcia. Predicting distributions of waiting times in customer service systems using mixture density networks. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–6. IEEE, 2019.
- [RTL21] Majid Raeis, Ali Tizghadam, and Alberto Leon-Garcia. Queue-learning: A reinforcement learning approach for providing quality of service. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 461–468, 2021.
- [SBCV⁺20] Alexey Strokach, David Becerra, Carles Corbi-Verge, Albert Perez-Riba, and Philip M Kim. Fast and flexible protein design using deep graph neural networks. *Cell systems*, 11(4):402–411, 2020.
- [SBV20] Jonathan Shlomi, Peter Battaglia, and Jean-Roch Vlimant. Graph neural networks in particle physics. *Machine Learning: Science and Technology*, 2(2):021001, 2020.
- [Sch81] Paul Schweitzer. Approximate analysis of multiclass closed networks of queues. *J. ACM*, 29(2), 1981.

- [SGT⁺08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.
- [She20] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.
- [Shi95] Leyuan Shi. Approximate analysis for queueing networks with finite capacity and customer loss. *European Journal of Operational Research*, 85(1):178–191, 1995.
- [SLA⁺22] Hamta Sedghani, Mina Zolfy Lighvan, Hadi S Aghdasi, Mauro Passacantando, Giacomo Verticale, and Danilo Ardagna. A stackelberg game approach for managing ai sensing tasks in mobile crowdsensing. *IEEE Access*, 10:91524–91544, 2022.
- [SLJ⁺15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
- [SM81] Kenneth C Sevcik and Isi Mitrani. The distribution of queueing network states at input and output instants. *Journal of the ACM (JACM)*, 28(2):358–371, 1981.
- [Smi90] Connie U Smith. *Performance engineering of software systems*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [Ste09] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton university press, 2009.
- [TB17] Kishor S Trivedi and Andrea Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.
- [TBBR13] Bernard W Taylor, CR Bector, SK Bhatt, and Earl Saul Rosenbloom. *Introduction to management science*. Pearson Boston, MA, USA, 2013.

- [Tho06] Nigel Thomas. Approximation in non-product form finite capacity queue systems. *Future Generation Computer Systems*, 22(7):820–827, 2006.
- [TRF03] Kishor S Trivedi, Srinivasan Ramani, and RICARDO Fricks. Recent advances in modeling response-time distributions in real-time systems. *Proceedings of the IEEE*, 91(7):1023–1037, 2003.
- [Tri12] Mirco Tribastone. A fluid model for layered queueing networks. *IEEE Transactions on Software Engineering*, 39(6):744–756, 2012.
- [TVS17] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*. CreateSpace Independent Publishing Platform, 2017.
- [VCC⁺17] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *Stat*, 1050(20):10–48550, 2017.
- [VWK⁺20] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *Acm computing surveys (csur)*, 53(2):1–33, 2020.
- [WCF22] Runan Wang, Giuliano Casale, and Antonio Filieri. Enhancing performance modeling of serverless functions via static analysis. In *International Conference on Service-Oriented Computing*, pages 71–88. Springer, 2022.
- [WCF23] Runan Wang, Giuliano Casale, and Antonio Filieri. Estimating multiclass service demand distributions using markovian arrival processes. *ACM Transactions on Modeling and Computer Simulation*, 33(1-2):1–26, 2023.
- [WF02] Murray Woodside and Greg Franks. Tutorial introduction to layered modeling of software performance, 2002.
- [WFP07] Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE’07)*, pages 171–187. IEEE, 2007.

- [Whi84] Ward Whitt. Open and closed models for networks of queues. *AT&T Bell Laboratories Technical Journal*, 63(9):1911–1979, 1984.
- [WNPM95] M Woodside, John E Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(1):20–34, 1995.
- [Woo21] Murray Woodside. Performance models of event-driven architectures. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 145–149, 2021.
- [WPC⁺20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.
- [WWS⁺23] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. Graphspd: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2409–2426. IEEE, 2023.
- [XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.
- [ZCP20] Lulai Zhu, Giuliano Casale, and Iker Perez. Fluid approximation of closed queueing networks with discriminatory processor sharing. *Performance Evaluation*, 139:102094, 2020.
- [ZW05] Tao Zheng and Murray Woodside. Fast estimation of probabilities of soft deadline misses in layered software performance models. In *Proceedings of the 5th International Workshop on Software and Performance*, pages 181–186, 2005.

- [ZZL⁺23] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.