ChainNet: A Customized Graph Neural Network Model for Loss-aware Edge AI Service Deployment

Zifeng Niu
Imperial College London
zifeng.niu19@imperial.ac.uk

Manuel Roveri
Politecnico di Milano
manuel.roveri@polimi.it

Giuliano Casale
Imperial College London
g.casale@imperial.ac.uk

***Abstract*—Edge AI seeks for the deployment of deep neural network (DNN) based services across distributed edge devices, embedding intelligence close to data sources. Due to capacity constraints at the edge, a difficult challenge lies in planning a dependable deployment that minimizes the data loss rate so as to meet application Quality-of-Service (QoS) goals. In this paper, we present ChainNet, a customized graph neural network (GNN) model serving as a surrogate to assess the reliability of alternative deployments and guide the loss-aware search for an optimal edge AI deployment plan. Extensive results show that ChainNet delivers a substantial improvement in loss prediction accuracy by over 50% compared to established GNN models, such as graph attention networks (GATs). Moreover, we show that ChainNet provides significantly more dependable deployment decisions under a fixed time budget compared to simulation-based search across a spectrum of systems from small to large-scale.**

***Index Terms*—graph neural networks, surrogate model, dependable loss-aware deployment**

I. Introduction

Deep neural network (DNN) models play a key role in a wide range of modern applications [1]–[5]. Since DNN models are resource-hungry, inference jobs have been initially run only in high-performance data centers [6]. However, in recent years, to increase the security and privacy of users [7], and also to improve energy efficiency and latency [8], an increasingly common approach is to rely on edge architectures that run inference jobs located near the end user that generates the data [9]. 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 [10]. 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 paper, 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 [9], 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 [11], 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 [12]. Yet, explicit solutions in the presence of multiple service chains and finite constraints are not known, due to the loss of product-form characteristics [13]. 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 [14]. 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 paper 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

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00034

make these contributions to address the deployment problem:

- We capture the underlying queueing structure of an edge AI system through ChainNet. Compared to popular GNN models, such as graph attention networks (GATs) [15], 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.
- We propose a detailed design that enables ChainNet to generalize its predictions to larger systems, even when faced with more intricate dependencies than those encountered during training. An ablation study demonstrates the effectiveness of our design, confirming its ability to achieve high generalization performance.
- ChainNet reduces the prediction error by over 50% compared to baseline GNN models such as GATs. With a fixed time budget, ChainNet is also found to deliver more reliable deployment decisions, reducing the data loss rate by an average of around 83% compared to decisions made by simulation-based search without resorting to a GNN. Extensive experiments show that ChainNet can support decision-making significantly 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 paper is organized as follows: the loss-aware deployment problem is formulated in Section II. A motivating example is presented in Section III. Background on GNNs is discussed in Section IV. ChainNet and its generalized design are developed in Section V and VI. The surrogate optimization program is illustrated in Section VII. Evaluation results are shown in Section VIII. Related work and conclusion are given in IX and X.

II. Problem Formulation

An edge AI service system comprises a set of D heterogeneous devices indexed by $k = 1, \ldots, D$. 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 $i = 1, \ldots, C$. 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 $j = 1, \ldots, T_i$, 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

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

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 $j = 1, \ldots, T_i$ 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.,

$$\begin{aligned} \max_{p \in \mathcal{P}} \quad & X_{total}(p) = \sum_{i=1}^{C} X_i(p) \\ \text{s.t.} \quad & \Delta m_k \leq M_k, \quad \forall k \in \{1, \ldots, D\}, \end{aligned} \tag{2}$$

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.

III. A Motivating Example

Queueing network (QN) models are a widely used performance modeling tool for AI and edge systems [16]–[18]. 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.

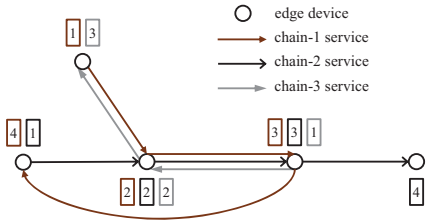


Fig. 1. An example of deploying three edge AI services on five devices. Both chain-1 and chain-2 services comprise four fragments, while chain-3 service comprises three fragments. Fragments are represented by boxes. The number within a box represents the position of this fragment in its service chain.

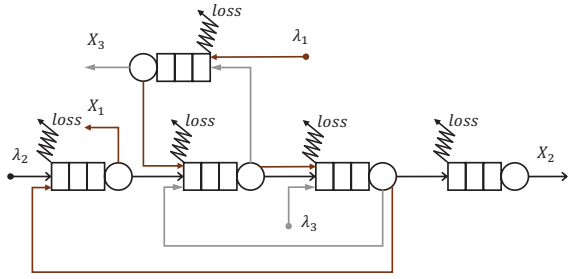


Fig. 2. The queueing network model of the placement in Fig. 1. A station with a finite-capacity buffer is represented by a circle followed by a series of rectangles. The arrival rates of requests for three services are λ_1, λ_2, and λ_3.

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 paper, consistently with this observation, we do not explicitly model the total network transmission time, only the performance at the devices.

Fig. 1 provides an example of a placement decision. The fragments of three service chains are executed by five 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 Fig. 2.

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 [19], 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 [20], [21], 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 [22]. Therefore, we explore building a learning-based surrogate model for the reliability assessment of candidate large-scale multi-chain placement decisions.

IV. Background on Graph Neural Networks

Graph neural networks (GNNs) are a class of neural networks designed to process data that is represented as a graph [23]. 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. The input data of a GNN model typically comprises a graph and a set of features associated to the nodes.

The main objective of GNN models is to learn a meaningful node embedding $\boldsymbol{h}_v$ for $v \in V$ on which regression or classification tasks can be built. An embedding is defined as the latent vector representation of a node that captures essential information of its own features and surrounding context. A popular paradigm to obtain $\boldsymbol{h}_v$ is message passing [24], where each node v in the graph (i) aggregates messages from its neighbors and (ii) updates its own embedding based on the obtained aggregate. The message-passing function is typically defined as

$$\boldsymbol{h}_v^{(n+1)} = \phi\Big(\boldsymbol{h}_v^{(n)}, f\Big(\Big\{\boldsymbol{h}_u^{(n)}\Big| u \in \mathcal{N}(v)\Big\}\Big)\Big), \tag{3}$$

where f and ϕ are respectively aggregation function and update function, n is an iteration index, and $\mathcal{N}(v)$ is the set of nodes adjacent to v. The choice of f, ϕ can impact the GNN model performance [25] and various options are possible. For instance, f could be an element-wise max-pooling, while ϕ may be a concatenation of $\boldsymbol{h}_v^{(n)}$ and f followed by a linear transformation.

Node embeddings are initialized with their input features. The message passing step (3) is iterated N times, which allows node embeddings to be iteratively updated and capture the important relationships within the data. After N iterations of message passing, we use $\boldsymbol{h}_v^{(N)}$ for node-level prediction tasks. In addition, it is possible to aggregate node embeddings to produce a representation of the entire graph for graph-level prediction tasks.

V. Methodology

A. Workflow

Fig. 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 overall system throughput according to (2). The central part of the optimizer is ChainNet, serving as a surrogate to estimate the performance of candidate placement decisions.

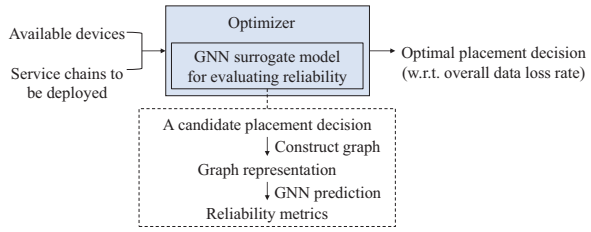


Fig. 3. Main steps of the proposed approach.

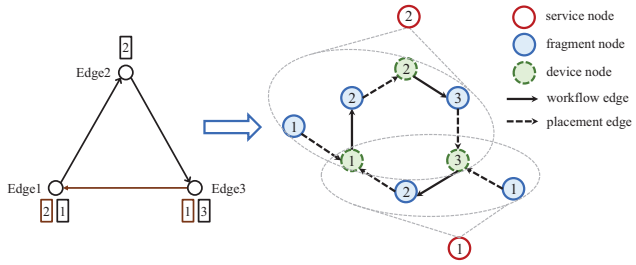


Fig. 4. Creating input graph corresponding to the placement decision. There are five types of elements in a graph representation: (i) service nodes: hollow circles with solid red borders; (ii) fragment nodes: filled nodes with solid blue borders; (iii) device nodes: filled nodes with dashed green borders (iv) workflow edges: solid line arrows; (v) placement edges: dashed line arrows.

B. 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 1). 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 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 II.

An example of graph construction is displayed in Fig. 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.

Algorithm 1 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**

Fig. 5. The execution sequence of the chain-i service comprising T_i fragments. It is partitioned into execution steps $E_1, \ldots, E_{T_i}$. The color scheme follows the one used in Fig. 4.

C. Design rationale based on queueing analysis

1) Graph partition: We partition the entire graph into a set of *execution steps*. An execution step is comprised of 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 Fig. 5, the execution sequence of the chain-i service is $E_1 \rightarrow \cdots \rightarrow E_{T_i}$, where E_j, $j = 1, \ldots, T_i$ 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.

2) 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 some arrivals get dropped when the buffer is full, the throughput typically shows a gradual decrease along the execution sequence. As a result, the system throughput is equivalent to 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.

3) 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 [26], the measures (i)-(iii) are interdependent under steady-state conditions. We leverage this mutual dependency in the design of message aggregation

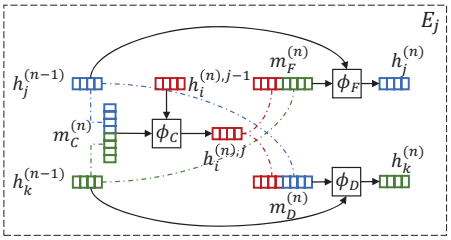


Fig. 6. The n-th iteration of the proposed message passing within the j-th execution step of the chain-i service. The color scheme follows the one used in Fig. 4. The colored dash-dotted lines copy node embeddings. The black arrows indicate input/output of the three update functions ϕ_C, ϕ_F, and ϕ_D.

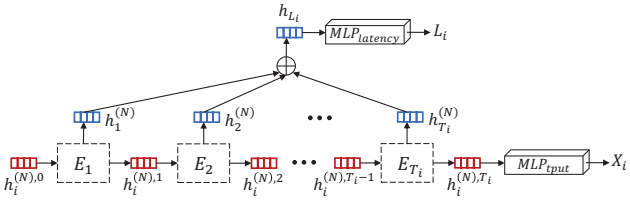


Fig. 7. Predicting system throughput and end-to-end latency of the chain-i service concurrently at the last (N-th) iteration. The $\oplus$ represents element-wise addition of vectors. The color scheme follows the one used in Fig. 4.

within each execution step by relating messages in patterns that resemble the interdependencies that the measures have among themselves.

D. 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 Fig. 6, three types of embeddings are involved in this process, which we now discuss in the next subsections.

1) 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, $j = 1, \ldots, T_i$ using an update function ϕ_C

$$\boldsymbol{h}_i^{(n),j} = \phi_C\left(\boldsymbol{h}_i^{(n),j-1}, \boldsymbol{m}_C^{(n)}\right), \tag{4}$$

where $\boldsymbol{m}_C^{(n)}$ 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.,

$$\boldsymbol{h}_i^{(n+1),0} = \boldsymbol{h}_i^{(n),T_i}. \tag{5}$$

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 $\boldsymbol{m}_C^{(n)}$, 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

$$\boldsymbol{m}_C^{(n)} = f\left(\left\{\boldsymbol{h}_j^{(n-1)}, \boldsymbol{h}_k^{(n-1)}\right\}\right). \tag{6}$$

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

$$\boldsymbol{h}_j^{(n)} = \phi_F\left(\boldsymbol{h}_j^{(n-1)}, \boldsymbol{m}_F^{(n)}\right). \tag{7}$$

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

$$\boldsymbol{m}_F^{(n)} = f\left(\left\{\boldsymbol{h}_i^{(n),j}, \boldsymbol{h}_k^{(n-1)}\right\}\right). \tag{8}$$

TABLE I
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
$\boldsymbol{h}_i$	service node embedding
$\boldsymbol{h}_j$	fragment node embedding
$\boldsymbol{h}_k$	device node embedding

3) Device node embedding: The state of the device is captured by its node embedding, which is updated once per iteration as well

$$\boldsymbol{h}_k^{(n)} = \phi_D\left(\boldsymbol{h}_k^{(n-1)}, \boldsymbol{m}_D^{(n)}\right). \tag{9}$$

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

$$\boldsymbol{m}_D^{(n)} = f\left(\left\{\boldsymbol{h}_i^{(n),j}, \boldsymbol{h}_j^{(n-1)}\right\}\right). \tag{10}$$

4) Message aggregation and update: In our framework, $\boldsymbol{m}_C^{(n)}$, $\boldsymbol{m}_F^{(n)}$, and $\boldsymbol{m}_D^{(n)}$ 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

$$f(\{\boldsymbol{x}, \boldsymbol{y}\}) = [\boldsymbol{x} \,||\, \boldsymbol{y}], \tag{11}$$

where $||$ 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 [27] as the three update functions. The weights of each GRU cell are shared across all execution steps.

E. Overall Algorithm

Based on previous developments, we summarize the algorithm of our GNN model (Algorithm 2).

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, 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, $\boldsymbol{h}_i^{(N),T_i}$, and all fragment nodes, $\boldsymbol{h}_1^{(N)}, \dots \boldsymbol{h}_{T_i}^{(N)}$, of every service chain. Following the design rationale in Section V-C2, 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 (line 17)

$$X_i = MLP_{tput}(\boldsymbol{h}_i^{(N),T_i}), \quad L_i = MLP_{latency}(\boldsymbol{h}_{L_i}), \tag{12}$$

where $\boldsymbol{h}_{L_i} = \sum_{j=1}^{T_i} \boldsymbol{h}_j^{(N)}$. This mechanism of prediction after completing the last (N-th) iteration is illustrated in Fig. 7.

For the training phase of our GNN model, the weights of five neural networks (ϕ_C, ϕ_F, ϕ_D, MLP_{tput} and $MLP_{latency}$) are initialized via the Glorot approach [28]. 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

$$L = \frac{1}{2Q} \sum_{i=1}^{Q} \Big((X_i - X_i^{gt})^2 + (L_i - L_i^{gt})^2 \Big), \tag{13}$$

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.

VI. Generalized GNN for Larger Placements

A. 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 $\boldsymbol{m}_D^{(n)}$

$$\boldsymbol{m}_D^{(n)} = f_{multi}\Big(\Big\{ \boldsymbol{m}_{D_1}^{(n)}, \dots, \boldsymbol{m}_{D_{F_k}}^{(n)} \Big\} \Big). \tag{14}$$

Algorithm 2 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 $\boldsymbol{h}_i^{(1),0}$, $\boldsymbol{h}_j^{(0)}$, $\boldsymbol{h}_k^{(0)}$
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 $\boldsymbol{m}_C^{(n)}$ by (6)
6: update service node embedding $\boldsymbol{h}_i^{(n),j}$ by (4)
7: produce fragment message $\boldsymbol{m}_F^{(n)}$ by (8)
8: update fragment node embedding $\boldsymbol{h}_j^{(n)}$ by (7)
9: **end for**
10: initialize $\boldsymbol{h}_i^{(n+1),0}$ for the next iteration by (5)
11: **end for**
12: **for** $k = 1, \dots, d$ **do**
13: produce device message $\boldsymbol{m}_D^{(n)}$ by (10)
14: update device node embedding $\boldsymbol{h}_k^{(n)}$ by (9)
15: **end for**
16: **end for**
17: obtain performance measures by (12)

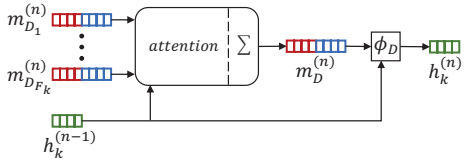


Fig. 8. Message passing for a device node shared by multiple execution steps. The color scheme follows the one used in Fig. 4.

Then, we use the aggregated message $\boldsymbol{m}_D^{(n)}$ to update the state of the shared device in the same way as (9).

For each involved execution step $t = 1, \dots, F_k$, we obtain its device message $\boldsymbol{m}_{D_t}^{(n)}$ using (10).

Next, we adopt a graph attention mechanism [29] to compute the attention score that characterizes the significance of a device message to the shared device

$$e(\boldsymbol{h}_k^{(n-1)}, \boldsymbol{m}_{D_t}^{(n)}) = \boldsymbol{\alpha}^T \sigma\Big(\boldsymbol{W} \Big[\boldsymbol{h}_k^{(n-1)} \,||\, \boldsymbol{m}_{D_t}^{(n)} \Big] \Big), \tag{15}$$

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

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

$$\alpha_{kt} = \frac{\exp(e(\boldsymbol{h}_k^{(n-1)}, \boldsymbol{m}_{D_t}^{(n)}))}{\sum_{u=1}^{F_k} \exp(e(\boldsymbol{h}_k^{(n-1)}, \boldsymbol{m}_{D_u}^{(n)}))}. \tag{16}$$

Using the normalized attention, the aggregated device message is finally obtained as a weighted sum of transformed device messages given by $f_{multi} = \sum_{t=1}^{F_k} \alpha_{kt} \boldsymbol{W} \boldsymbol{m}_{D_t}^{(n)}$.

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

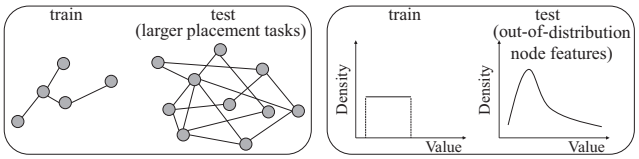


Fig. 9. Two out-of-distribution scenarios considered by our surrogate model.

on the attention mechanism so that the impact of different execution steps on the shared device can be considered. This adjustment is shown in Fig. 8.

B. 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 Fig. 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.

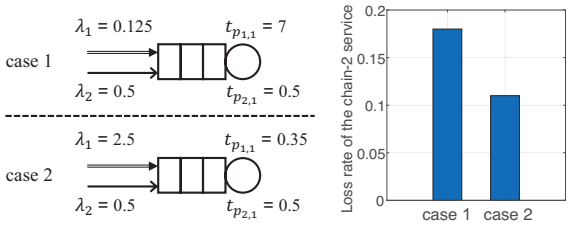
1) Larger placement problems: The proposed GNN model needs to generalize well to larger placement problems beyond training samples. Compared to the training samples, they could have more service chains, more fragments within a service chain, or more available devices.

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 path.

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 $\boldsymbol{h}_{L_i} = \frac{1}{T_i}\sum_{j=1}^{T_i} \boldsymbol{h}_j^{(N)}$.

2) Out-of-distribution node features: As described in Section V-B, 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 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.



(a) (b)

Fig. 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.

TABLE II
FEATURE MODIFICATIONS FOR GENERALIZATION

	GNN output		$\boldsymbol{h}_i^{(1),0}$	$\boldsymbol{h}_j^{(0)}$			$\boldsymbol{h}_k^{(0)}$
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.
$\Delta t_k = \sum_{i=1}^{C}\sum_{j=1}^{T_i} p_{i,j,k} t_{p_{i,j}}$

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, Fig. 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, Δ_{t_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 II.

VII. SURROGATE OPTIMIZATION METHOD

The proposed surrogate optimization algorithm seeks to maximize the constrained program that we have introduced in (2). The underpinning search strategy relies on simulated annealing (SA) [30]. 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 (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 (2).

TABLE III
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.

TABLE IV
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 [32]
learning rate α	0.001, decay 10% per 10 epochs

TABLE V
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

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.

VIII. 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 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.

A. Experiment setup

1) 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 [31]. Each JMT simulation abstracts the real deployment in a way similar to Figure 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 III, 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 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 generalization ability of the model.

B. Evaluating the proposed GNN model

1) 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 performance measure and ground truth, respectively. Because we predict the performance of every service chain in a collection of network samples,

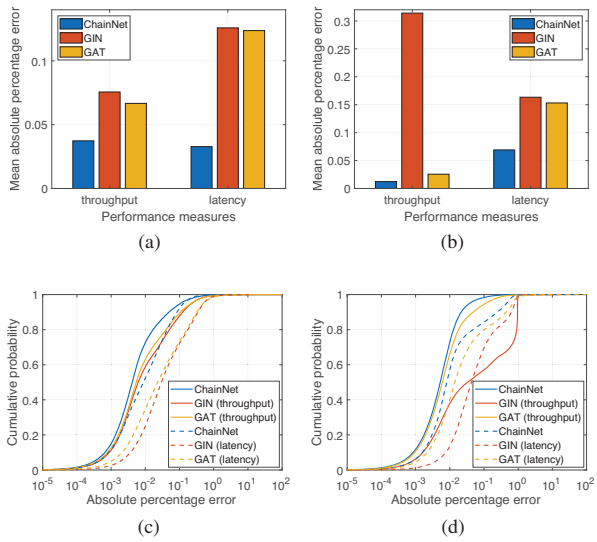


Fig. 11. (a)-(b): MAPE results on Type I and II test datasets, respectively. (c)-(d): APE distributions on Type I and II test datasets, respectively.

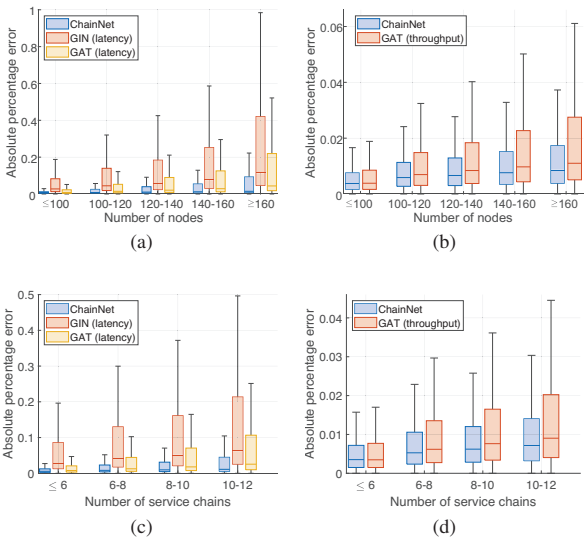


Fig. 12. Box plot of APE distributions under different number of nodes and service chains. The horizontal line that splits the box is the median. We only present the throughput prediction error boxes of ChainNet and GAT in Fig. 12b and Fig. 12d. This is because the medians of GIN boxes are higher than the third quartiles of other boxes, making it inconvenient to display them alongside ChainNet and GAT.

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:

$$\text{MAPE} = \frac{100\%}{Q_{test}} \sum_{i=1}^{Q_{test}} \left| \frac{P_i - G_i}{G_i} \right|, \tag{17}$$

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

2) Baseline: We compare our customized GNN model against two baseline models: graph isomorphism network (GIN) [25] and graph attention network (GAT) [15], which are two widely used GNN models adopted for different tasks arising in various application domains of AI [33]–[36]. 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. The message-passing for nodes is performed within their respective neighbor groups. Motivated by DNN chains in edge AI services, ChainNet partition 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. IV.

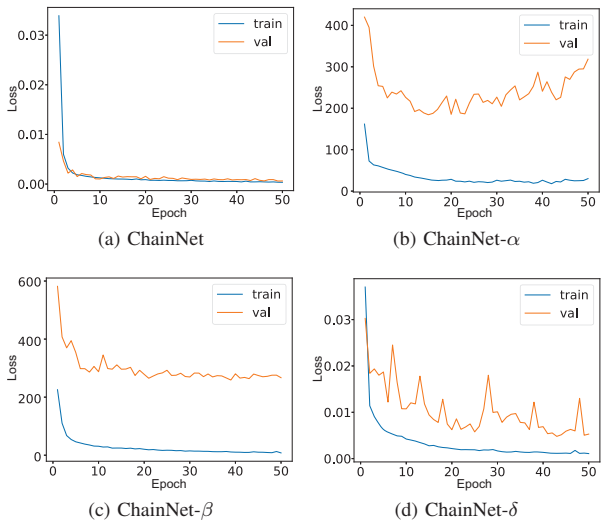
3) Results: Fig. 11 shows the MAPE and APE distributions for the predicted system throughput and end-to-end latency in Type I and II test datasets. Compared to the best results of the baseline models, ChainNet delivers a substantial error reduction of 48.0% and 64.2% on throughput and latency, respectively. The percentiles of the APE distribution on throughput predictions are shown in Table V. In the table, the value at the ith 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 II. In Table V, we also provide the results of baseline models when using original features in Table II, 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 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 Fig. 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.

4) Ablation study: The generalization method proposed in Table II consists of two modifications: correcting (i) GNN

TABLE VI
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

Fig. 13. Ablation study. Variants α, β, and δ do not use the modifications in Table II. A major degradation appears in their validation losses supporting the evidence that removal of GNN features degrades ChainNet.

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 II; (ii) ChainNet-β: without the modification of the GNN outputs in Table II; (iii) ChainNet-δ: without the modifications for the input $\boldsymbol{h}_i^{(1),0}$, $\boldsymbol{h}_j^{(0)}$, and $\boldsymbol{h}_k^{(0)}$ in Table II. The MAPE given by ChainNet and its three variants is shown in Table VI. 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. Fig. 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.

TABLE VII
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.

C. Evaluating the surrogate optimization program

1) placement problems: In this section, we solve 100 randomly generated placement problems, each corresponding to solving an optimization program (2), using our surrogate method developed in Section VII. 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 VII.

2) 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 VII 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.

Fig. 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.

3) Performance metric: We define the loss probability of a placement p as

$$\pi_{loss}(p) = \frac{\lambda_{total} - X_{total}(p)}{\lambda_{total}}, \tag{18}$$

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

$$\eta(p) = \frac{\pi_{loss}(p_0) - \pi_{loss}(p)}{\pi_{loss}(p_0)} = \frac{X_{total}(p) - X_{total}(p_0)}{\lambda_{total} - X_{total}(p_0)}, \tag{19}$$

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.

4) 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:

a) 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.

b) 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 search steps to 30 trials for both GNN and simulator, thus comprising 3,000 steps in total. As the GNN is a surrogate of the simulated data, it is expected that an accurate GNN model should display very similar performance 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.

5) Results: Given a placement problem, the ChainNet-based optimization program provides us with an optimized placement 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 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.

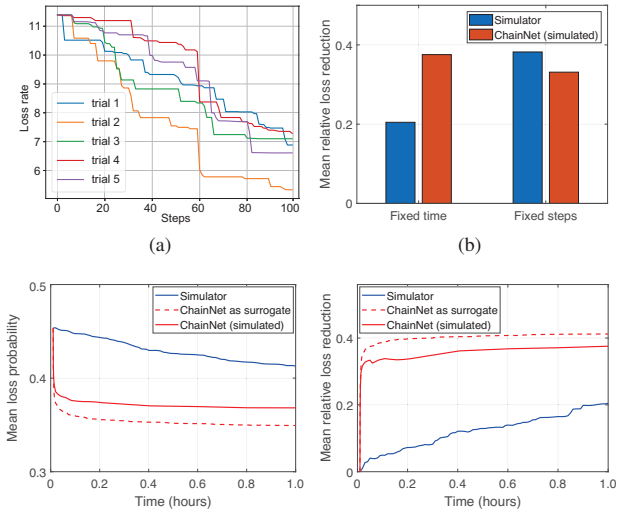


Fig. 14. (a): An example of different objective function reductions across 5 independent trials. (b): Mean relative loss reduction by ChainNet and baseline method in two groups of experiments. (c)-(d): The change process of mean loss probability and mean relative loss reduction over the fixed time frame. 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.

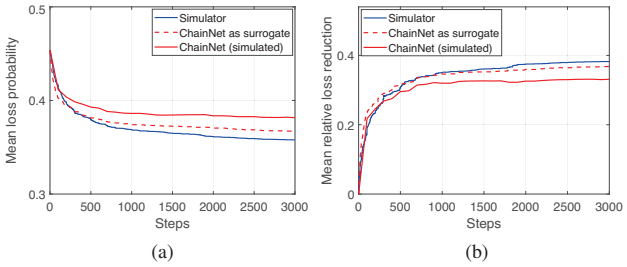


Fig. 15. (a)-(b): The change process of mean loss probability and mean relative loss reduction over the same number of steps. The curves indicate that ChainNet is accurate in capturing trends of simulation-based search.

a) Overall results: Fig. 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 search steps, ChainNet achieves 86.7% of the level of the baseline method, with a high time efficiency.

b) 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 (Fig. 14c to 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.

c) Fixed-steps optimization: For this scenario, we plot the change of the two metrics over the same number of steps (Fig. 15a to 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.

D. 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×OrangePi Zero, 2×Raspberry Pi A+, and 1×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 [37]. Each model is partitioned into three and four fragments, forming two distinct service chains. Thus the problem has a total of 2×4=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 reinforce the generalization ability of ChainNet.

IX. Related Work

Several studies seek to optimize the deployment of DNN models across distributed edge devices. The deployment can be optimized by considering various metrics, such as latency [38]–[41], throughput [42], [43], energy consumption [44], and accuracy [10]. These studies can be classified into two main groups. The first focuses on the partition of DNN models, which determines the split points that produce near-optimal model fragments [10], [38], [42], [43], [45], [46]. The second pursues the near-optimal placement decision for the given DNN models and partitioned fragments [39]–[41], [44]. For the second group, 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. In complex scenarios with queues and loss, deriving the exact mathematical expression for the objective function is often challenging, necessitating the use of surrogate to bridge this gap. Graph learning-based models appear as promising surrogates in graph-structured problems [47]. Although ordinary GNNs can be used conveniently, to obtain desirable performance in specific problems, the model typically requires a 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 [48]–[52].

X. Conclusion, Limitations and Future Work

In this paper, we propose ChainNet, a novel customized GNN that can be used as a surrogate for loss-aware edge AI deployment. The design of this model is guided by queueing analysis tailored to the scenario where congestion and loss occur. Meanwhile, it has a strong generalization ability that enables ChainNet to perform well in complicated problems. In addition, ChainNet can predict throughput and latency concurrently because of its customized design. The evaluation results demonstrate that ChainNet has distinct advantages compared with baseline models, in terms of accuracy and generalization capability, which shows promise in evaluating and settling complex edge deployment. It effectively and efficiently drives the search for a deployment that minimizes the data loss rate.

The proposed method has two main limitations. Firstly, the DNN models considered in this paper 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 forward execution, such as custom early-exit networks [53] or directed acyclic graph (DAG) [6], [54]. To accommodate these scenarios, an extension of ChainNet may incorporate Markovian routing, which would need to go beyond the determinism of ChainNet. Secondly, network link unreliability could be integrated in extensions in ChainNet, such as unsuccessful transmissions caused by link failure [55]. This would also be achievable once Markovian routing is supporting, so as to enable probabilistic routing of jobs on failed links to a sink node modeling their loss.

In terms of future work, given the inference speed shown in the case study, the proposed ChainNet-based optimization program may be expanded to other tasks outside design, such as real-time resource management scenarios, possibly integrating it with online optimization [56], [57]. It would also be interesting to compare it with other classes of GNNs, such as those with implicit layers [58], which have not been applied before in Edge AI deployment modeling.

REFERENCES

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

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

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[4] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019, pp. 4171–4186.

[5] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[6] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 330–339.

[7] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," pp. 37–42, 2015.

[8] M. Kaiser, R. Griessl, N. Kucza, C. Haumann, L. Tigges, K. Mika, J. Hagemeyer, F. Porrmann, U. Rückert, M. vor dem Berge *et al.*, "Vedliot: very efficient deep learning in iot," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 963–968.

[9] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[10] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[11] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[12] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[13] N. Thomas, "Approximation in non-product form finite capacity queue systems," *Future Generation Computer Systems*, vol. 22, no. 7, pp. 820–827, 2006.

[14] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," 2017.

[15] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio *et al.*, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48 550, 2017.

[16] Q. Liang, W. A. Hanafy, A. Ali-Eldin, and P. Shenoy, "Model-driven cluster resource management for ai workloads in edge clouds," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 18, no. 1, pp. 1–26, 2023.

[17] H. Sedghani, M. Z. Lighvan, H. S. Aghdasi, M. Passacantando, G. Verticale, and D. Ardagna, "A stackelberg game approach for managing ai sensing tasks in mobile crowdsensing," *IEEE Access*, vol. 10, pp. 91 524–91 544, 2022.

[18] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, vol. 20, no. 5, pp. 1907–1923, 2020.

[19] H. Baumann and W. Sandmann, "Multi-server tandem queue with markovian arrival process, phase-type service times, and finite buffers," *European Journal of Operational Research*, vol. 256, no. 1, pp. 187–195, 2017.

[20] L. Shi, "Approximate analysis for queueing networks with finite capacity and customer loss," *European journal of operational research*, vol. 85, no. 1, pp. 178–191, 1995.

[21] F. Ciucu, F. Poloczek, and A. Rizk, "Queue and loss distributions in finite-buffer queues," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 2, pp. 1–29, 2019.

[22] H. Kobayashi and B. L. Mark, "Generalized loss models and queueing-loss networks," *International Transactions in Operational Research*, vol. 9, no. 1, pp. 97–112, 2002.

[23] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[24] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[25] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*.

[26] L. JD C, "A proof of the queuing formula: L= λw," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[27] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[28] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[29] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *International Conference on Learning Representations*.

[30] A. G. Nikolaev and S. H. Jacobson, "Simulated annealing," *Handbook of metaheuristics*, pp. 1–39, 2010.

[31] M. Bertoli, G. Casale, and G. Serazzi, "Jmt: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] B.-H. Kim and J. C. Ye, "Understanding graph isomorphism network for rs-fmri functional connectivity analysis," *Frontiers in neuroscience*, vol. 14, p. 630, 2020.

[34] T. Alsinet, J. Argelich, R. Béjar, D. Gibert, and J. Planes, "Argumentation reasoning with graph isomorphism networks for reddit conversation analysis," *International Journal of Computational Intelligence Systems*, vol. 15, no. 1, p. 86, 2022.

[35] K. Fountoulakis, A. Levi, S. Yang, A. Baranwal, and A. Jagannath, "Graph attention retrospective," *Journal of Machine Learning Research*, vol. 24, no. 246, pp. 1–52, 2023.

[36] H. Hou, S. Ding, X. Xu, and L. Ding, "A novel clustering algorithm based on multi-layer features and graph attention networks," *Soft Computing*, vol. 27, no. 9, pp. 5553–5566, 2023.

[37] G. Casale and M. Roveri, "Scheduling inputs in early exit neural networks," *IEEE Transactions on Computers*, 2023.

[38] E. Cho, J. Yoon, D. Baek, D. Lee, and D.-H. Bae, "Dnn model deployment on distributed edges," in *International Conference on Web Engineering*. Springer, 2021, pp. 15–26.

[39] M. Bensalem, J. Dizdarević, and A. Jukan, "Modeling of deep neural network (dnn) placement and inference in edge computing," in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2020, pp. 1–6.

[40] G. Constantinou, C. Shahabi, and S. H. Kim, "Placement of dnn models on mobile edge devices for effective video analysis," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 207–218.

[41] S. Disabato, M. Roveri, and C. Alippi, "Distributed deep convolutional neural networks for the internet-of-things," *IEEE Transactions on computers*, vol. 70, no. 8, pp. 1239–1252, 2021.

[42] A. Parthasarathy and B. 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)*. IEEE, 2022, pp. 239–246.

[43] J. Li, W. Liang, Y. Li, Z. Xu, X. Jia, and S. 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.

[44] G. Premsankar and B. Ghaddar, "Energy-efficient service placement for latency-sensitive applications in edge computing," *IEEE internet of things journal*, vol. 9, no. 18, pp. 17 926–17 937, 2022.

[45] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco, "Distributed inference acceleration with adaptive dnn partitioning and offloading," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 854–863.

[46] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "Deepslicing: collaborative and adaptive cnn inference with low latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[47] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
[48] J. Shlomi, P. Battaglia, and J.-R. Vlimant, "Graph neural networks in particle physics," *Machine Learning: Science and Technology*, vol. 2, no. 2, p. 021001, 2020.
[49] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2409–2426.
[50] M. Ferriol-Galmés, J. Paillisse, J. Suárez-Varela, K. Rusek, S. Xiao, X. Shi, X. Cheng, P. Barlet-Ros, and A. Cabellos-Aparicio, "Routenet-fermi: Network modeling with graph neural networks," *IEEE/ACM Transactions on Networking*, 2023.
[51] M. Réau, N. Renaud, L. C. Xue, and A. M. Bonvin, "Deeprank-gnn: a graph neural network framework to learn patterns in protein–protein interfaces," *Bioinformatics*, vol. 39, no. 1, p. btac759, 2023.
[52] S. Huang, Y. Wei, L. Peng, M. Wang, L. Hui, P. Liu, Z. Du, Z. Liu, and Y. Cui, "xnet: Modeling network performance with graph neural networks," *IEEE/ACM Transactions on Networking*, 2023.
[53] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *International Conference on Machine Learning*. PMLR, 2017, pp. 527–536.
[54] X. Dai, Z. Xiao, H. Jiang, M. Lei, G. Min, J. Liu, and S. Dustdar, "Offloading dependent tasks in edge computing with unknown system-side information," *IEEE Transactions on Services Computing*, 2023.
[55] A. Aral and I. Brandić, "Learning spatiotemporal failure dependencies for resilient edge computing services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1578–1590, 2020.
[56] S. Modaresi, D. Sauré, and J. P. Vielma, "Learning in combinatorial optimization: What and how to explore," *Operations Research*, vol. 68, no. 5, pp. 1585–1604, 2020.
[57] S. Padakandla, "A survey of reinforcement learning algorithms for dynamically varying environments," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–25, 2021.
[58] J. Liu, B. Hooi, K. Kawaguchi, and X. Xiao, "Mgnni: Multiscale graph neural networks with implicit layers," in *Advances in Neural Information Processing Systems*, vol. 35. Curran Associates, Inc., 2022, pp. 21 358–21 370.