BCEdge: SLO-Aware DNN Inference Services with Adaptive Batch-Concurrent Scheduling on Edge Devices

Ziyang Zhang, *Student Member, IEEE*, Yang Zhao, *Senior Member, IEEE*, Huan Li, *Senior Member, IEEE*, and Jie Liu, *Fellow, IEEE*

***Abstract*—As deep neural networks (DNNs) are increasingly used in a broad spectrum of edge intelligent applications, it is often necessary to provide multi-DNN model inference services, and it is nontrivial for edge inference platforms to simultaneously deliver high-throughput and low-latency. Such edge devices with multi-DNN model pose new challenges for scheduler designs. First, edge devices should be capable of efficiently scheduling multiple heterogeneous DNN models in order to optimize system utilization. Second, each inference request may have different service level objectives (SLOs) to improve quality of service (QoS). To address these challenges, this paper proposes BCEdge, a novel learning-based scheduling framework that incorporates adaptive batching and concurrent execution of DNN inference services on edge devices. We first propose a shared memory policy to reduce the memory contention among multiple DNN models. Afterwards, a utility function is defined to evaluate the trade-off between throughput and latency. The scheduler in BCEdge leverages branch-based deep reinforcement learning (DRL) to maximize utility by 1) optimizing batch size, 2) automatically identifying the number of concurrent instances for multiple DNN models, and 3) determining the shared memory configuration among multiple DNN models. Besides, the lightweight DNN-based prediction model in BCEdge can achieve SLO awareness by reducing the performance interference among multiple DNN models. Our prototype implemented on various edge devices illustrates that BCEdge enhances utility by up to 37.6% and reduces memory usage by up to 38% on average, compared to state-of-the-art schemes, while maintaining the SLO violation rate within 5%.**

***Index Terms*—Edge Computing, Inference Service, Scheduling, Reinforcement Learning, Service Level Objective (SLO).**

I. Introduction

DNN inference service systems deployed on cloud servers provide multiple trained deep neural networks (DNNs) for users. These systems are usually multi-tenant, meaning that each DNN model has one or more concurrent instances to serve various inference applications, while making full use of the abundant computing resources on servers. For instance, the multi-instance GPU (MIG) in NVIDIA Ampere architecture enables the partitioning of a single NVIDIA A100 GPU into seven independent GPU concurrent instances. These concurrent instances can achieve DNN model inference in parallel, enabling the GPU to achieve up to 7× utilization with a guaranteed quality of service (QoS). Nevertheless, concurrently executing multiple instances of different DNN models results in complex interference between DNN models. In addition, prior work has adopted batching to process requests with lower inference cost [1], [2], [3]. Batching refers to the aggregation of multiple requests into a single batch request, with a given time window [4]. DNN inference service systems process one batch request at a time, thereby improving system throughput ($e.g.$, measured in requests per second, rps). Although system throughput can be improved by increasing the batch size, there is a challenge, larger batch size leads to longer waiting time for requests to be processed, which inevitably increases latency.

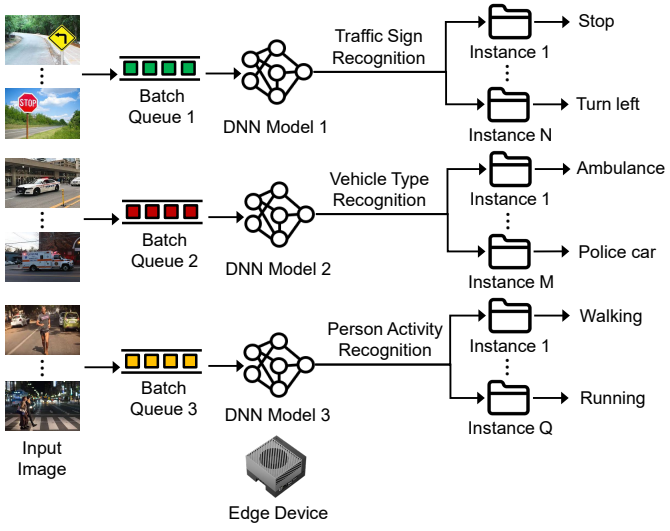


Fig. 1. Batching and concurrent inference services on edge devices with video surveillance application in autonomous driving as an example.

For edge inference serving systems, computility and memory enhancements provide new opportunities for efficiently deploying DNN inference systems on edge accelerators ($e.g.$, graphics processing unit (GPU), tensor processing unit (TPU),

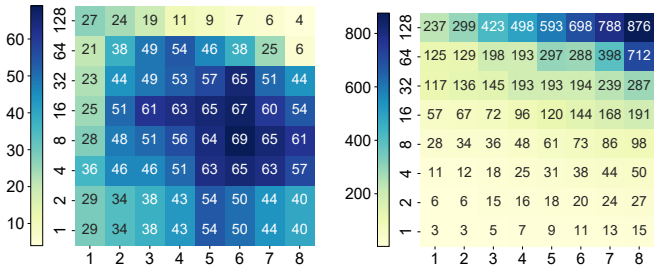
Ziyang Zhang is with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang 150006, China. E-mail: {zhangzy,lincy}@stu.hit.edu.cn

Yang Zhao, Huan Li and Jie Liu are with the International Research Institute for Artificial Intelligence, Harbin Institute of Technology, Shenzhen, Guangdong 518071, China. E-mail: {yang.zhao,huanli,jieliu}@hit.edu.cn.

Manuscript received 11 Sep 2023; revised 01 Feb 2024; accepted 02 Jun 2024.

This work is partly supported by the National Key R&D Program of China under Grant No. 2021ZD0110905, No. 2022YFF0503900, and An Open Competition Project of Heilongjiang Province, China, on Research and Application of Key Technologies for Intelligent Farming Decision Platform, under Grant No. 2021ZXJ05A03.

(Corresponding authors: Yang Zhao, Jie Liu.)



(a) Throughput (rps) (b) Latency (ms)

Fig. 2. The effects of batching and concurrent inference on (a) system throughput and (b) end-to-end latency. Throughput is measured as requests-per-second (rps). The x-axis represents the number of instances of DNN model (denotes as c_i), and the y-axis represents the batch size (denotes as b_i). We use YOLO-v5 [12] on NVIDIA Xavier NX edge device with 8GB DRAM.

and vision processing unit (VPU), $etc.$). There has been a tremendous amount of research effort using various DNN model lightweight techniques ($e.g.$, model pruning [5], knowledge distillation [6], low-bit quantization [7]), aiming at optimizing the inference latency and throughput of edge inference services. Batch inference for multiple concurrent requests, however, is also urgent in practice for edge inference services that not much research effort has focused on.

As shown in Fig. 1, we take the video surveillance application deployed on edge devices in autonomous driving [8] as an example to illustrate the use case scenario of batch inference on edge devices. To achieve full-scene awareness on the road, the video surveillance application uses multiple DNN models, including traffic sign recognition [9], vehicle type recognition [10], and person activity recognition [11], to batch video streams captured by on-board cameras in real-time. Furthermore, each DNN model has multiple instances to concurrently process video frames with different objectives. In order to guarantee low-latency inference services while maintaining throughput performance, it is necessary to investigate how batching and the number of concurrent instances affect the latency and throughput performance of inference requests.

Motivating Example: to better understand the performance impact of batching and concurrent inference, we perform an experimental study using YOLO-v5 [12] on NVIDIA Xavier NX edge device. Due to resource constraints on edge devices, we leverage TensorRT [13] to accelerate the original DNN model. Fig. 2 reports throughput and latency for different batch sizes and numbers of concurrent instances. An interesting observation is that high throughput is achieved when the batch size (denoted as b_i) and the number of concurrent instances (denoted as c_i) both have intermediate values. For example, when we run YOLO-v5 model on NVIDIA Xavier NX, the edge inference serving system achieves the highest throughput (69 rps), when the batch size $b_i = 8$, and the number of concurrent instances $c_i = 6$, as shown in Fig. 2(a). The reason why the throughput has diminishing returns as b_i and c_i increase is due to resource contention caused by interference [14] among multiple DNN models. In addition, the effects of the batch size and the number of concurrent instances on the inference latency are also investigated and shown in Fig. 2(b), which shows different behaviors from the throughput. From Fig. 2, we see that when the batch size and the number of concurrent instances have excessively high values, $e.g.$, $b_i = 128$, $c_i = 8$, the throughput is significantly reduced and the latency is extremely high. Therefore, it is critical to design an efficient scheduler to 1) co-optimize throughput and latency, and 2) predict the interference between multi-DNN model for an edge DNN inference serving system.

Challenges: designing such an edge inference service system must address the following challenges. First, inference requests have latency service level objective (denoted as SLO_L^i) $i.e.$, a bounded response latency (for instance, an interactive application typically requires a response time less than 100 milliseconds [4], [15], [16]), to achieve quality of service (QoS) with low-latency. It is necessary to consider how to ensure the QoS of inference requests to avoid SLO violation. SLO violation means that the end-to-end latency of the inference request exceeds the bounded response latency. Second, edge devices usually deploy multi-DNN model to improve system throughput and resource utilization. The interference between multi-DNN model needs to be considered to improve system robustness. To this end, we propose BCEdge, a learning-based, adaptive, multi-tenant scheduling framework for SLO-aware DNN inference services at the edge. First, a shared memory policy in BCEdge can effectively reduce memory usage in order to avoid resource contention among multiple DNN models. Second, a lightweight neural networks-based interference prediction model is used to achieve SLO-aware, thereby alleviating interference between multi-DNN model. Finally, a learning-based scheduler leverages the branching architecture to automatically determine the appropriate batch size, number of concurrent instances for multiple DNN models, and shared memory configuration among multiple DNN models, in order to achieve the optimal trade-off between throughput and latency.

Table I provides a summarized comparison of BCEdge with prior work. All prior work involves adaptively adjusting the batch size, either automatically or manually, to achieve high throughput. Although some prior studies have investigated multi-DNN model, they did not support the scheduling of concurrent instances of homogeneous multiple DNN models, which becomes increasingly important to fully utilize the computing resources on edge devices. In addition, only TF-Serving [17] considers model interference. Except for Clipper [1] and DeepRT [15], none of the other works consider strict latency budget. In contrast, our proposed BCEdge addresses all of the above issues.

To evaluate the BCEdge framework, we implement the system and build a prototype. We evaluate the BCEdge prototype on five heterogeneous edge devices with six representative DNN models widely used in both computer vision and natural language processing applications, as shown in Table IV. Evaluations show that BCEdge improves the trade-off between throughput and latency by 10%∼62% compared to the state-of-the-art (SOTA) schemes while reducing memory usage by up to 62%. Moreover, the SLO violation rate of BCEdge is kept within 5%.

The main contributions of this paper are as follows:

- We propose BCEdge, a learning-based scheduling frame-

TABLE I
COMPARISON WITH PRIOR WORK

Service Framework	Adaptive Batching	Concurrent Instance	Shared Memory	SLO Aware
TF-Serving [17]	✓	✗	✗	✗
Triton [3]	✓	✓	✓	✗
Clipper [1]	✓	✗	✗	✓
Prema [18]	✓	✗	✓	✓
DVABatch [19]	✓	✗	✓	✗
Gpulet [4]	✓	✓	✗	✓
DeepRT [15]	✓	✓	✗	✓
Ďelen [16]	✗	✓	✓	✗
BATCH [2]	✓	✗	✗	✓
BCEdge (Ours)	✓	✓	✓	✓

work for adaptive batching and concurrent DNN inference service on edge devices. This framework is motivated by a real-world case study on the effects of batching and concurrent inference of DNN models on throughput and latency performance on edge devices.

- We leverage a branch-based deep reinforcement learning algorithm to automatically adjust the batch size, the number of concurrent instances for multiple DNN models, and the shared memory configuration among multiple DNN models to co-optimize throughput and latency.
- We propose a shared memory policy to reduce memory contention among multiple DNN models. In addition, the lightweight DNN-based prediction model reduces the performance interference between multi-DNN model.
- Extensive experiments on implemented system prototypes show that BCEgde outperforms state-of-the-art schemes in improving the throughput-latency trade-off, reducing SLO violation rate and memory usage.

The rest of the paper is organized as follows: Section II presents related work. Section III describes system model and problem formulation. Section IV illustrates our framework design in detail. Section V describes our framework implementation and experimental setup. Section VI reports evaluation results. Section VII discusses the limitations and future work of our proposed framework. Section VIII concludes our work.

II. RELATED WORK

A. On-Device Model-Level DNN Inference Service

On-device DNN inference services using single computing platforms have been extensively studied recently. According to the level of DNN inference optimization, it can be divided into coarse-grained model level and fine-grained operator level. In particular, the on-device model-level DNN inference service aims to optimize the entire DNN model using a single computing platform. Prior work treats DNN model as an indivisible scheduling unit, and propose a series of inference serving frameworks to provide DNN inference services [1], [2], [4], [15], [17], [19], [20], [21], [22], [23]. Clipper [1], TensorFlow-Serving [17], MArK [20], DeepRT [15], and BATCH [2] adopt conventional adaptive batching that use time window for DNN inference. Ďelen [16] uses early exit mechanism for DNN models to achieve fine-grained scheduling of inference requests. None of these existing frameworks offer concurrent operation of model instances to further improve throughput. There is also prior work that focuses on SLO-aware DNN inference service. Gpulet [4] leverages spatio-temporal sharing of computing resources for multiple heterogeneous DNN models with latency constraints. Clockwork [23] exploits predictable execution times to achieve tight SLO. INFaaS [22] reduces cost and SLO violation, as well as improves throughput by choosing adequate model variant. PSLO [21] is a preempting SLO-aware scheduler based on minimum average expected latency, which aims to trade-off system throughput and SLO. Unlike the above work, we foucs on reducing the SLO violation rate ($i.e.$, the proportion of inference requests that exceed SLO) caused by the interference of multi-DNN model. In addition, some edge inference frameworks involve privacy protection [24], [25] and edge-cloud collaborative [26], [27], respectively. These works are complementary to BCEdge that can alleviate privacy and resource constraints.

B. On-Device Operator-Level DNN Inference Service

Deep learning frameworks ($e.g.$, TensorFlow and PyTorch) abstract a particular DNN model as a directed acyclic graph (DAG) when executing DNN inference [28]. DAG consists of nodes and edges, and the nodes represent operators (various operations in the DNN model), such as convolution, pooling, batch normalization, activation, $etc.$ Edges represent dependencies between operators. The on-device operator-level DNN inference services aim to improve QoS by optimizing the operators of the DNN model using a single computing platform. Except for model-level inference services, prior work also focuses on optimizing the operator-level scheduling of DNN models to enhance the QoS of model service [14], [18], [29], [30], [31]. REEF [29] adopts a parallel mechanism based on dynamic kernel padding to improve throughput. VELTAIR [14] proposes an adaptive operator-level compilation and scheduling framework to achieve efficient resource usage, and reduces interference-induced performance loss. PREMA [18] proposes a predictive multi-task scheduling algorithm to achieve high-throughput. Abacus [30] leverages overlap-aware latency prediction model and deterministic scheduling of overlapped DNN operators to improves throughput while maintaining QoS. These works are also complementary to BCEdge that enable higher throughput and lower latency.

C. DNN Inference Service in Edge-Cloud Collaboration

On-device DNN inference service is difficult due to resource constraints on edge devices. To this end, prior work leverages edge-cloud framework to facilitate collaborative inference services. Edge-Cloud collaboration executes inference by reasonably allocating sub-models of DNN between cloud servers and edge devices [32]. This computing paradigm enables DNN inference to provide high QoS in a real-time responsive manner. TVW-RL [33] exploits various temporal resource usage patterns of time-varying workloads using a deep reinforcement learning (DRL) approach, to improve utilization in real production traces. Likewise, KaiS [34], A3C-R2N2 [35], MILP [36], A3C-DO [37], and MFRL [38] proposed different multi-agent DRL-based scheduling algorithms, to optimize throughput, latency, energy consumption, cost, $etc.$

TABLE II
NOTATIONS

Notation	Description
x_i	The i-th inference request
m_i	The i-th DNN model type
d_i	Input data shape for i-th inference request
SLO_L^i	Latency service level objective of the i-th inference request
t_i	The i-th scheduling time slot
b_i	Batch size for i-th DNN model
c_i	Number of concurrent instances for i-th DNN model
s_i	Shared memory configuration among multiple DNN models
U	Utility function
D_i	The input data size of i-th inference request
R_i	The output data size of i-th inference request
$\mathcal{B}$	Network bandwidth between IoT devices and edge devices
C_i	The computility of edge devices
f_i	The clock frequency of edge devices
l_{end}^i	End-to-end latency for i-th inference request
T_i	Throughput at i-th time slot t_i
o_i	memory usage for i-th inference request
e_i	Parameter space for i-th DNN model
g_i	Runtime space for i-th DNN model
s_t	State in DRL at time slot t_i
a_t	Action in DRL at time slot t_i
r_t	Instant reward at time slot t_i
F_b	Features of batch size
F_c	Features of concurrent instances
F_s	Features of shared memory configuration

MCDS [39] leverages a tree-based search strategy and a DNN-based prediction model to optimize QoS. Similar to MILP [36], DeEdge [40] proposes D-Deads, a distributed greedy scheduling algorithm with task-deadline, which maximizes throughput while minimizing latency. Note that the above work only schedule individual tasks one by one, ignoring the benefits of batching and concurrent inference. Inspired by above work, BCEdge can be extended to an edge-cloud collaborative inference framework to further optimize specific objectives.

III. System Model and Problem Formulation

In this section, the system models including the DNN inference request model, the service scheduling model and the end-to-end latency model are formulated and discussed. Afterwards, an optimization problem is formulated to include the effects of the batch size, the number of concurrent instances for multiple DNN models and shared memory configuration. Table II provides the key notations used in this paper.

A. System Models

Request Model: suppose there are multiple IoT devices ($e.g.$, cameras, drones, smartphones, $etc.$) sharing the resources of edge devices. Before task scheduling, IoT devices generate a series of inference requests with DNN model type m_i, input data shape d_i, and latency service level objective SLO_L^i. The i-th inference request x_i, therefore, can be denoted as $x_i = \{m_i, d_i, SLO_L^i\}$. BCEdge maintains a batch of requests queue for each DNN model, and supports dynamic batching by aggregating multiple inference requests with same DNN model into corresponding batch request queue. Meanwhile, BCEdge creates multiple instances for each DNN model ($e.g.$, concurrent instances c_i), since it is critical for edge devices with GPUs to leverage both batching and concurrent inference to improve throughput. The model zoo in BCEdge backend executes DNN inference from batch request queue, and returns prediction result.

Scheduling Model: since the SLO is different for each request, a fixed scheduling time slot is inappropriate. In addition, it is impractical to specify scheduling time slot for each request individually, which significantly increases system overhead, $i.e.$, the scheduling latency at runtime. Therefore, we set the i-th scheduling time slot t_i as the ratio of the sum of SLO_L^i for a batch of requests to the number of concurrent instances for multiple DNN models, which can be denoted as:

$$t_i = \sum_{i=1}^{b_i} SLO_L^i / c_i \tag{1}$$

where b_i is the batch size, and c_i is the number of concurrent instances for multiple DNN models. In this way, BCEdge is enabled to guarantee the SLO of each request, and provides efficient inference services via batching and concurrent inference. Note that BCEdge starts the next scheduling immediately after finishing the current scheduling to reduce the GPU idle.

End-to-end latency Model: end-to-end latency involves the communication time between IoT devices and edge devices, as well as model inference time, which consists of the following components:

- *request transmission time* l_{trans}^i: the time that IoT devices send the i-th inference request to edge devices via the network, which depends on network bandwidth and input data size:

$$l_{trans}^i = \frac{D_i}{\mathcal{B}}, \tag{2}$$

  where $\mathcal{B}$ is the network bandwidth between IoT devices and edge devices. D_i is the input data size of i-th inference request.
- *request queuing time* l_{que}^i: the time that a request is blocked on the request queue until it is scheduled. We utilize the $M/M/1$ queuing model to define the queuing time of inference requests. Assume that the arrivals of requests are independent of each other, and the interval between arrival times follows a Poisson distribution with parameter λ. The service time of requests follows a negative exponential distribution with parameter μ, and the number of servers is 1. Therefore, the average queuing time of a request can be modeled as follows:

$$l_{que}^i = \frac{\rho}{\mu - \lambda}, \tag{3}$$

  where $\rho = \mu/\lambda$ is the service intensity per unit time.
- *DNN inference time* l_{infer}^i: the time that edge device executes model inference, which depends on input data size, the computility and clock frequency of edge devices, batch size, as well as number of concurrent instances.

$$l_{infer}^i = \frac{D_i \cdot C_i}{f_i \cdot b_i \cdot c_i}, \tag{4}$$

  where C_i is the computility of edge devices. f_i is the clock frequency of edge devices.

- *result response time* l^i_{rsp}: the time that edge devices transmit results to IoT devices via the network, which depends on network bandwidth and output data size:

$$l^i_{rsp} = \frac{R_i}{\mathcal{B}}, \tag{5}$$

where R_i is the data size of the return result for the i-th inference request.

Overall, the end-to-end latency l^i_{end} can be denoted as:

$$l^i_{end} = l^i_{trans} + l^i_{que} + l^i_{infer} + l^i_{rsp} \tag{6}$$

System Throughput Model: system throughput, as one of the SLO, refers to the number of requests that the inference service system can process within a time window (*e.g.*, measured in requests per second, rps). Increasing throughput means more inference requests can be processed within the same time window. In our work, system throughput (denoted as rps) depends on batch size (b_i), the number of concurrent instances for multiple DNN models (c_i), and end-to-end latency (l^i_{end}), which can be formulated as follows:

$$rps \propto \frac{b_i \cdot c_i}{l^i_{end}} \tag{7}$$

where $\propto$ means that system throughput (rps) is proportional to batch size (b_i), the number of concurrent instances for multiple DNN models (c_i), and inversely proportional to end-to-end latency (l^i_{end}).

B. Problem Formulation

Our objective is to co-optimize both throughput and latency for each DNN model by automatically exploring the feasible set of batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models while guaranteeing SLO. Inspired by the co-adaptive scheduler named Pollux [41], we propose a *utility function* to evaluate the trade-off between throughput and latency:

$$U(T_i, L_i) = log(T_i(b_i, c_i, s_i) / \frac{L_i(b_i, c_i, s_i)}{(\sum_{i=1}^{b_i} SLO^i_L)}) \tag{8}$$

where s_i indicates the shared memory configuration among multiple DNN models (see Section IV-D for details). The throughput of the i-th scheduling time slot t_i can be denoted as $T_i(b_i, c_i, s_i)$, and $L_i(b_i, c_i, s_i)$ is the actual end-to-end latency of the i-th scheduling time slot t_i. $\sum_{i=1}^{b_i} SLO^i_L$ denotes the ratio of the sum of SLO^i_L for a batch of requests to the number of concurrent instances for multiple DNN models.

Meanwhile, inference service system must consider the memory capacity of edge devices, denoted as O_i, as well as the latency constraints. Therefore, the optimization objective is formulated as follows:

$$\begin{aligned} &\max_{b_i, c_i, s_i} \ U(T_i, L_i) \\ &s.t. \ o_i \leq O_i \\ &L_i \leq \sum_{i=1}^{b_i} SLO^i_L \end{aligned} \tag{9}$$

where o_i is the memory usage of the i-th inference request.

IV. System Design

A. System Overiew

We propose BCEdge, an adaptive batching and concurrent inference service framework for edge devices. The scheduler in BCEdge aims to allocate a moderate batch size, number of concurrent instances, and shared memory configuration among multiple DNN models for each inference request, while maintaining the SLO boundary. Unlike prior work that only considers a subset of three dimensions [1], [15], we propose a scheduler that fully explores three dimensions to better tradeoff throughput and latency.

Fig. 3 shows an overview of proposed scheduling framework, namely BCEdge, which consists of dynamic batching (Section IV-B), concurrent instance (Section IV-C), shared memory for multiple DNN models (Section IV-D), learning-based scheduler (Section IV-E), and SLO-aware interference predictor (Section IV-F). BCEdge first ❶ maintains a batch request queue for each DNN model. The requests with different DNN models generated by IoT devices are merged to send the corresponding batch request queues. The performance profiler ❷ periodically collects information (*e.g.*, resource utilization, system throughput and end-to-end latency) for each DNN model. Meanwhile, the SLO-aware interference predictor ❸ analyzes the potential interference caused by concurrent instances between multiple DNN models, which guides the scheduler to make more robust decisions. The learning-based scheduler then ❹ identifies the optimal batch size, the number of concurrent instances for multiple DNN models, and shared memory configuration among multiple DNN models by leveraging profiled information. Finally, the runtime engine ❺ executes concurrent DNN inference service with batching on edge devices.

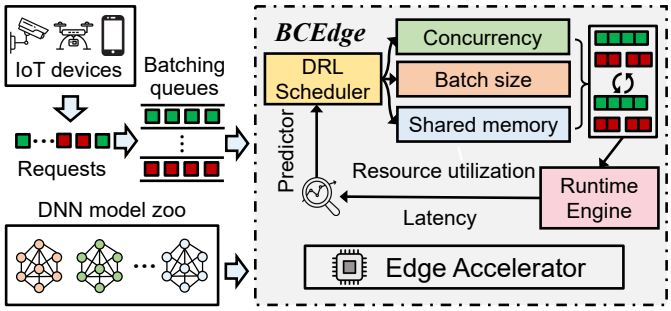


Fig. 3. Overview of the proposed scheduling framework for DNN inference services.

B. Dynamic Batching

BCEdge enables dynamic batch inference by allowing a single request to specify a batch of inputs. Inference for a batch of inputs is executed concurrently, which is important for GPUs as it can improve throughput significantly. As Fig. 4 shows, dynamic batching first maintains a batch request queue separately for each DNN model. Dynamically created batches then are dispatched to all concurrent instances configured for each DNN model, and multiple batch request queues are concurrently executed. More precisely, dynamic batching adds each request to the corresponding batch request queue based

on the order of arrival. Meanwhile, it sorts the priority of each inference request based on SLO, the shorter the SLO, the higher the priority. Dynamic batching aggregates multiple requests into one large request, and dispatches the batch requests to multiple slots of each DNN model at runtime. Note that each slot is a model instance. Additionally, the batch requests are scheduled in the order of arrival if they have the same priority.

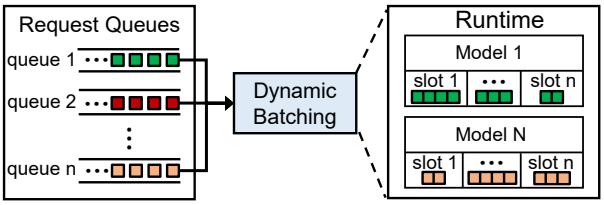


Fig. 4. Illustration of dynamic batching.

C. Model Instance Concurrency

BCEdge enables multi-DNN model and multi-instance of the same DNN model to execute in parallel, and the DNN models executed on CPU are handled similarly by BCEdge. Fig. 5 shows the pipeline of three DNN models executing multi-instance in parallel, and each DNN model creates three instances. Suppose BCEdge is not processing any requests currently. When the first three requests arrive at the same time (one for each model), each instance of the three DNN models will process a corresponding request. BCEdge then dispatches these concurrent instances to GPU immediately, and the hardware scheduler starts processing the corresponding inferences in parallel. Note that the first three inference requests are immediately executed in parallel, while subsequent inference requests have to wait until the corresponding previous requests are completed.

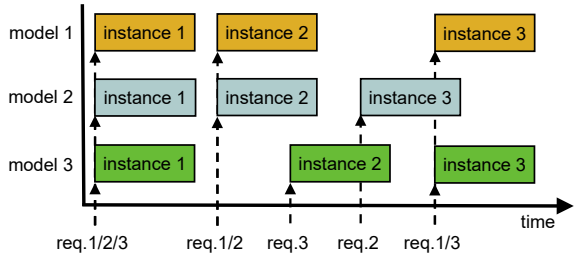


Fig. 5. Illustration of concurrent instance.

D. Shared Memory Policy for Multiple DNN Models

DNN model inference is computationally intensive, which requires high-overhead GPU memory space, including the following four components: uncontrollable space, user data, model parameters, and runtime space. Specifically, the uncontrollable space refers to the space allocated by the operating system, such as the memory space occupied by the context of each process in the hardware accelerator. User data is the user-specified memory space, such as the input and output of the DNN model. Model parameters refer to the memory space occupied by the parameters of the DNN model, which need to be loaded into GPU memory for inference. The runtime space is the memory occupied by the operators of the DNN model during calculation.

In multi-DNN model concurrent inference service, the inevitable memory contention among multiple DNN models can cause serious performance interference. Importantly, unlike cloud server-level GPUs, edge GPUs are resource-constrained and lack effective memory management mechanisms. To address these challenges, we propose a shared memory policy for multiple DNN models to reduce memory usage. As mentioned before, the number of concurrent instances for i-th DNN model is c_i. The parameter space and runtime space of each DNN model are denoted as $e_i(i = 1, 2, \cdots, E)$ and $g_i(i = 1, 2, \cdots, G)$, respectively. Note that the uncontrollable space is automatically allocated by the operating system, which cannot be modified. In addition, the memory space occupied by user data is negligible. Thus, uncontrollable space and user data are not considered.

The shared memory policy for multiple DNN models s_i can be formalized as follows:

$$s_i = \begin{cases} e_i + g_i * c_i, & i = 1 \\ \sum_{i=1}^{M} e_i + \max(g_i * c_i), & i > 1 \end{cases} \tag{10}$$

where i is the number of DNN models. The first case considers multi-instance concurrency for a single model ($i.e.$, $i = 1$). For instance, model m_1 creates x instances, the parameter e_1 of DNN model m_1 only needs to be allocated once, and g_1 is shared by these x instances. The second case considers multi-instance concurrency for multiple DNN models ($i.e.$, $i > 1$). For instance, there are three DNN models, and the computing pipeline is $m_1 \rightarrow m_2 \rightarrow m_3$. Among them, model m_1 creates 2 instances, model m_2 creates 3 instances, and model m_3 creates 4 instances. The total memory usage, therefore, is $e_1 + e_2 + e_3 + \max(g_1 * 2, g_2 * 3, g_3 * 4)$. In this way, we can effectively reduce memory usage to avoid memory contention.

E. Learning-based Scheduler

Compared with traditional heuristic approaches, deep reinforcement learning (DRL) has great advantages in processing complex decision problems, which can be applied to search spaces with high-dimensional. Thus, the problem, introduced in Eq. (9), can be converted to a DRL problem to solve. The agent in DRL generates actions (decisions) by continuously interacting with the environment, and the interaction process is usually modeled by a Markov decision process (MDP). An MDP consists of the following state, action, transition probability and reward.

- ***State***: the state s reflects the state characteristics of the environment, and all states constitute the state space $\mathcal{S}$ ($s \in \mathcal{S}$). At each scheduling time slot t_i, the agent in DRL constructs a state $s_t(s_t \in \mathcal{S})$ to periodically collect request information and the resource utilization of edge devices. s_t consists of a three-tuple: (I) the DNN model type m_i. (II) the input date shape d_i. (III) the SLO of each requests SLO_L^i.
- ***Action***: the action a is the behavior taken by the agent, and all actions constitute the action space $\mathcal{A}$ ($a \in \mathcal{A}$). The action of the agent aims to identify optimal batch size b_i,

the number of concurrent instances for multiple DNN models c_i, and available shared memory configuration among multiple DNN models s_i, given specific DNN model m. Thus, the action $a_t (a_t \in \mathcal{A})$ at scheduling time slot t_i can be denoted as $a_t = (b_i, c_i, s_i)$.

- ***Transition probability***: the transition probability p is the probability distribution of the agent transitioning from the current state s to the next timestamp state s', satisfying $\sum_{s' \in \mathcal{S}} p(s'|s,a) = 1$.
- ***Reward***: the reward is a scalar value that the environment feeds back to the agent after the agent performs an action according to the policy π, which is related to current state, current action and the state of the next timestamp. The agent in DRL aims to maximize the accumulated expected reward $\mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$, where r_t is the instant reward at time slot t_i. The objective of our BCEdge design is to maximize the proposed utility in Eq. (8) that achieves the optimal trade-off between throughput and latency. Therefore, we propose a novel reward function $r_i(t)$ for each request i as follows:

$$r_i(t) = \begin{cases} U(T_i, L_i), & L_i \leq \sum_{i=1}^{b_i} SLO_L^i, o_i \leq O_i \\ e^{-L_i \cdot o_i}, & otherwise \end{cases} \tag{11}$$

where the agent in DRL is triggered by SLO violation. This is because the reward function in Eq. (11) depends on SLO violation. The first case indicates that when the latency of the current inference request is within the specified latency constraints ($i.e.$, SLO_L^i) and the memory usage does not exceed memory capacity ($i.e.$, O_i), we calculate the reward based on the inference latency of the request and the system throughput, and the reward function corresponds to the utility function in Eq. (8). For the second case, when the latency and/or memory of the current inference request exceeds, we use an exponential function to define the reward, which decreases as the inference latency increases.

The purpose of DRL is to identify an optimal policy $\pi^* = \text{argmax}_a Q^*(s,a)$ to maximize the long-term cumulative expected return by using the optimal Q-function (Q^*) to quantify this reward

$$Q^*(s,a) = \mathbb{E}_{\tau \sim p(\tau)}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{12}$$

where the Q-function $Q(s,a)$ is the reward achieved by executing action a at state s, which can leverage neural network, such as deep Q-network (DQN) [42], to approximate the optimal Q-function $Q^*(s,a)$, and γ is a discounting factor.

Search Space Challenge: Note that the scheduling in BCEdge is more complex compared with prior work ($e.g.$, TF-Serving [17], Clipper [1], and DeepRT [15]), since it involves batching, concurrent inference, and shared memory configuration multiple DNN models. Therefore, the trade-off configuration would sit on the sweet spot in the search space built upon the three dimensions, which creates a huge search space. However, we find that it is inefficient to directly apply traditional DRL to the huge three dimensionsal scheduling space in BCEdge. We suppose that M DNN models exit and each DNN model m has b_i batch size, c_i the number of concurrent instances for multiple DNN models, and s_i shared memory configuration among multiple DNN models. In total, $\coprod_{m=1}^{M} b_i \cdot c_i \cdot s_i$ different combinations (actions) need to be considered at each timestamp. Searching for an optimal policy in such a huge action space is time-consuming, which is unacceptable for online inference services on edge devices.

Branch DRL-based scheduling algorithm: A key insight to address this problem is that through an efficient scheduling strategy design. We propose to decouple the reward exploration for each scheduling dimension, so that we can reduce the complexity of the action space via parallel scheduling. In this way, the output dimension can be reduced from the *multiplication* $\coprod_{m=1}^{M} b_i \cdot c_i \cdot s_i$ to *summation* $\sum_{m=1}^{M} (b_i + c_i + s_i)$. In BCEdge, we employ a branch-based structure [43] to achieve a learning-based scheduler.

As Fig. 6 depicts, we first use a shared feature extractor implemented by a fully connected neural layer with 256 neural units, $i.e.$, the larger gray trapezoidal in Fig. 6, to extract common features F between different dimensions of the input state. Then, a fully connected neural layer with 64 neural units ($i.e.$, the smaller gray trapezoid in Fig. 6) is used for evaluation of the state value. Meanwhile, these common features are then fed into different feature extraction branches implemented by a fully connected neural layer with 64 neural units, $i.e.$, the smaller gray trapezoidal in Fig. 6, to generate a series of features (F_b, F_c, F_s), representing the features of batch size, the number of concurrent instances for multiple DNN models, and shared memory configuration among multiple DNN models, respectively. Each branch corresponds to a dimension in the three dimensions action space. In addition, the common features serve as an additional branch to estimate the state value function (V-function) of the state, where V-function is the expectation of $Q(s,a)$ about action a. The state value (V-value) is aggregated with the features of different branches to output the rewards of each branch. Finally, the optimal joint estimate of the distributed actions is obtained. Such parallel design essentially copes the trade-off between latency and optimality of agent exploration.

We extend Branching Dueling DQN [43] into our proposed scheduling algorithm. Similar to Dueling DQN [44], Branching Dueling DQN [43] is also an approximation to the optimal Q-function, which decomposes the optimal Q-function $Q^*(s,a)$ into an optimal V-function V^*, where $V^*(s) = \sum_{a \in \mathcal{A}} \pi(a|s_t) \cdot Q(s_t, a)$, and an optimal advantage function A^*, where $A^* = Q^*(s,a) - V^*(s)$, aiming at improving the learning efficiency of DQN [42].

Formally, for an action with d dimension $(d \in 1, 2, \cdots, n)$, the Eq. (12) transforms as follows:

$$Q_d^*(s, a_d) = V^*(s) + (A_d^*(s, a_d) - \frac{1}{n} \sum_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)) \tag{13}$$

where $|A_d|$ indicates n discrete sub-actions.

We alleviates the overestimation of DQN by leveraging two networks ($i.e.$, Q-network and target Q-network), which can be formulated as follows:

$$y_d = r_d + \gamma \frac{1}{n} \sum_{d} Q_d^-(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_d(s', a'_d)) \tag{14}$$

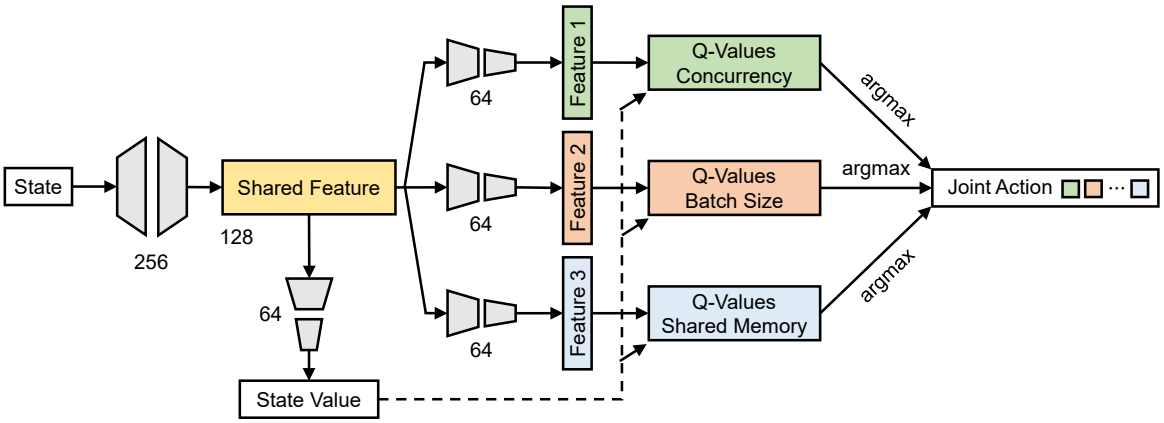


Fig. 6. Illustration of proposed branch DQN-based scheduler design with the parallel branches to determine different attributes. The gray trapezoids indicate the weights of the fully connected neural layers and the size of each layer (i.e. number of neural units) is indicated.

where y_d is the target Q-value with dimension d, and $Q_d(s', a'_d)$ is the Q-value estimated by neural network.

We train Branching Dueling DQN by aggregating distributed temporal difference (TD) errors across branches. The loss function can be formulated as follows:

$$\mathcal{L} = \mathbb{E}_{(s,a,r,s_{t+1})\sim\mathcal{D}}[\frac{1}{n}\sum_{d}(y_d - Q_d(s, a_d))^2] \quad (15)$$

Algorithm 1 provides the overall procedure of scheduling. The scheduler first receives the information of DNN model and resource utilization for each inference request. Before each scheduling time slot, all networks are initialized. For each scheduling time slot, the scheduler first checks each request queue. If the request queue is empty, it pushes incoming requests into the request queue (*line 7*). The branch network proposed in Fig. 6, is used to extract the features of each dimension and calculate the respective Q-value (*line 9∼11*). The scheduler selects a joint action $a_t(b_i, c_i, s_i)$ based on policy π with probability ϵ-greedy (*line 12*). Afterwards, is executes action a_t and observes reward using Eq. (8) (*line 13*). Meanwhile, the state changes from s_t to s_{t+1}, and the current state, action, reward and the next state are stored as a action transition in the replay buffer $\mathcal{D}$. The batch request queue are pulled from the batching slot when the batch requests are completed (*line 16*). Next, we calculate the Q-Value and target Q-Value of each dimension separately by random sampling (*line 17∼19*). The scheduler updates the parameters of all networks, and repeat the above process (*line 21∼24*) until the end of the iteration. Finally, the scheduler outputs the optimal batch size, the number of concurrent instances for multiple DNN models, and shared memory configuration among multiple DNN models.

Algorithm 1: Learning-based scheduling algorithm

Input : The specific information (m_i, d_i, SLO_L^i) of each inference request x_i
Output: batch size b_i, the number of concurrent instances for multiple DNN models c_i and shared memory configuration among multiple DNN models

1 s_i Initialization Q-network Q with random weights θ;
2 Initialization target Q-network $\hat{Q}$ with weight $\theta^- = \theta$;
3 Initialize an empty replay buffer $\mathcal{D} \leftarrow \emptyset$;
4 **for** *each scheduling time slot* t_i **do**
5 **for** *each environment step* **do**
6 **if** *request queue is empty* **then**
7 Push request x_i to batch request queue;
8 **end**
9 Use a shared feature extractor to extract common features F of the input state;
10 F derived features (F_b, F_c, F_s) for each dimension via different feature extraction branches;
11 F is aggregated with (F_b, F_c, F_s) for each dimension to output the Q-value for each dimension;
12 Select a random joint-action tuple $a_t(b_i, c_i, s_i)$ based on policy π with probability ϵ-greedy;
13 Execute action a_t and observe reward $r_t\,(a_t \mid s_t)$ using Eq. (11);
14 $s_{t+1} \sim p\,(s_{t+1} \mid s_t, a_t)$;
15 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r\,(s_t, a_t)\,, s_{t+1})\}$;
16 Pull current request queue from batching slot s_i;
17 Sample random minibatch of transitions $(s_t, a_t, r(s_t, a_t))$ from $\mathcal{D}$;
18 Calculate the target Q-value y_d using Eq. (14);
19 Calculate the branch Q-value $Q_d(s', a'_d)$ using Eq. (13);
20 **end**
21 **for** *each gradient step* **do**
22 Update the parameters of Q-network parameter θ using Eq. (15);
23 Update the parameters of target Q-network $\theta^- \leftarrow \theta$ every T steps;
24 **end**
25 **end**

F. SLO-Aware Interference Predictor

Concurrent inference of multi-DNN model or multi-instance of a single model can process more requests simultaneously to improve throughput. However, an important challenge is the performance interference caused by multi-DNN model concurrent inference. As shown in Fig. 2, we observed that concurrent inference significantly increases latency compared to executing a single model independently, as multi-DNN model compete for the shared resources on edge devices, especially the memory. In this case, model interference may causes the scheduler to make incorrect schedules, and possibly violates SLO.

The key to mitigating interference is accurately predicting latency increase when multi-DNN model inference are executed concurrently on a single GPU. Fig. 2 reveals that the number of concurrent for YOLO-v5 [12] model has a nonlinear relationship with the end-to-end delay, the prediction

model based on linear regression, therefore, cannot accurately achieve interference-aware. To confine the interference effect, we leverage a lightweight two-layer neural network (NN) with negligible overhead as the interference prediction model, which directly learns the interference latency of multi-DNN model concurrent inference on a single GPU. As shown in Fig. 7, the simple yet effective interference prediction model based on NN leverages historical latency and memory utilization as the input of the neural network. The interference prediction model then compares the estimated latency of the neural network output with the actual latency provided by the performance profiler in BCEdge. The neural network is trained by minimizing the standard deviation between actual latency and estimation latency, which aims to improve the stability of the scheduler and reduce the SLO violation rate.

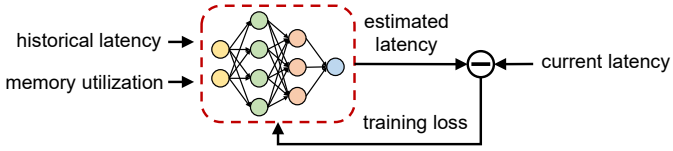


Fig. 7. SLO-aware interference predictor based on neural network.

V. Implementation

BCEdge Prototype: we implement the prototype of BCEdge using the runtime backend of Triton [3], a inference serving system provided by NVIDIA. Table III reports the detailed description of the evaluated inference system and the GPU specifications used. The table also provides the versions of operating system, CUDA, runtime, and deep learning (DL) framework. As Fig. 8 shows, we use two IMX cameras and a micorphone as IoT devices. Poisson distribution is used to model the arrival of events for many applications. For generating a realistic request arrival pattern, we sample inter-arrival time for each model from a Poisson random distribution, which are widely used to evaluate DNN inference serving services, based on previous literature [4], [14], [29], [45], [46]. The request arrival rate is set to 30 requests per second (rps), which are randomly distributed across all instances of each DNN workload. Unless otherwise indicated, all evaluations are reported on an NVIDIA Xavier NX edge GPUs.

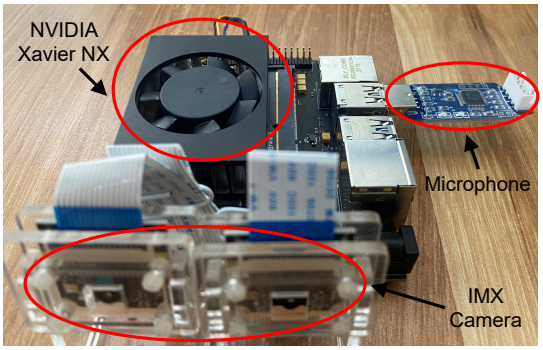


Fig. 8. BCEdge prototype implemented on NVIDIA Xavier NX edge devices. We use two IMX cameras and a micorphone as IoT devices.

DNN Workloads: we use six representative DNN models from three popular DNN families to process image and speech data. Specifically, we use YOLO-v5 [12] for object detection tasks, MobileNet-v3 [47], ResNet-18 [48], EfficientNet-B0 [49] and Inception-v3 [50] for image classification tasks, and TinyBERT [51] for speech recognition tasks. Additionally, we use NVIDIA TensorRT [13], a high-performance DNN inference optimization library for better batching and concurrent inference. The input shape, SLO and accuracy of each DNN model is listed in Table IV.

TABLE III
The evaluated system specifications

Edge device	NVIDIA Jetson Xavier NX
Operating system	Ubuntu: 18.04.6 (kernel 4.15.0)
Software	CUDA 10.2 and TensorRT 8.2 [13]
CPU	6-core Carmel@1.4GHz ARMv8.2 64-bit
GPU	384-core Volta GPU with 48 Tensor Cores
DRAM	8GB 128-bit LPDDR4x 59.7GB/s
Runtime	NVIDIA Triton Inference Server 2.19.0 [3]
DL framework	PyTorch 1.10

TABLE IV
List of DNN models used in the evaluation

Model	Input Shape	SLO (ms)	Accuracy (%)
YOLO-v5 (yolo)	3x224x224	138	44.8
MobileNet-v3 (mob)	3x224x224	86	67.3
ResNet-18 (res)	3x224x224	58	72.4
EfficientNet-B0 (eff)	3x224x224	93	77.1
Inception-v3 (inc)	3x224x224	66	76.5
TinyBERT (bert)	1x14	114	78.4

Training Details: our proposed Algorithm 1 is based on the branching dueling DQN [43] framework. All networks are trained using the Adam optimizer with a learning rate of 10^{-4}. Each network has a two-layer ReLU neural network with 128 and 64 hidden units, respectively, and the replay buffer size is set to 10^6. The target Q-network is updated every 10^3 time steps. We train it offline on a workstation using four NVIDIA GeForce GTX 3080 GPUs with a mini-batch size of 512 and a discount factor $\gamma = 0.99$ for 500 epochs, and deploy the trained algorithm online to edge devices for evaluation.

Baselines: As baselines, we compare against three DNN inference service systems, all listed in Table I. The baselines are composed by combining different approaches ($i.e.$, adaptive batching, concurrent instance, shared memory, and SLO aware). As discussed in Section I, existing DNN inference service systems mainly focus on enabling cloud-based inference service instead of providing edge inference service as BCEdge does. Consequently, we compare BCEdge against DeepRT [15] and Ďelen [16], which enable DNN inference service at the edge, to evaluate the effectiveness of our proposed approach. In addition, we also use BATCH [2] in Table I as a baseline, which is a cloud-based DNN inference serving framework that enables adaptive batching and SLO awareness. For fair comparison, we deploy DeepRT [15],Ďelen [16], and BATCH [2] on the same edge devices as the BCEdge environment. Details for the three baselines are as follows:

- ***DeepRT*** [15]: a soft real-time scheduler with dynamic batching using earliest-deadline-first (EDF) scheduling algorithm at the edge. We extend DeepRT to enable TinyBERT DNN model for speech recognition tasks.

- ***Ďelen*** [16]: a flexible and adaptive DNN inference service system for multi-tenant edge devices. Ďelen leverages multi-exit DNNs, $i.e.$, a mechanism that enables early exit at different points during DNN inference, to achieve fine-grained control over inference requests.
- ***BATCH*** [2]: a DNN inference service framework with adaptive batching using an optimizer based on an analytical model to provide SLO guarantees.

VI. Evaluation

A. Evaluation Metrics

We use our proposed utility function defined in Eq. (8), overall throughput, end-to-end latency, and SLO violation rate as our evaluation metrics.

- ***Utility***: this metric measures the trade-off between overall throughput and end-to-end latency, which represents the behavior of BCEdge that tends to improve throughput while ensuring SLO.
- ***Overall Throughput***: this metric represents how many inference requests from users can BCEdge serve on the target edge device ($i.e.$, measured in requests per second, rps).
- ***End-to-end Latency***: this metric measures the execution time for each inference request with latency constraints ($i.e.$, SLO).
- ***SLO violation Rate***: this metric measures the proportion of inference requests that exceed SLO within a time window.

B. Overall Performance

We first evaluate the trade-off performance ($i.e.$, the proposed utility in Eq. (8)) of BCEdge in terms of throughput and latency. Fig. 9 reports the normalized utility of six DNN models in Table IV. It can be observed that our proposed BCEdge consistently outperforms all baselines for all DNN models. Specifically, the utility of BCEdge is 48%, 27% and 18% higher than DeepRT, BATCH and Ďelen on average, respectively. The lower-utility of both DeepRT and BATCH is due to the lack of concurrent inference, although they leverage adaptive batching to improve throughput. Since the multi-exit mechanism in Ďelen can improve throughput while reducing latency, it has a better trade-off between throughput and latency than batching and concurrent inference. In contrast, BCEdge, on the one hand, fully explores the three dimensions search space consisting of batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models by efficient scheduling using branch DRL-based. On the other hand, the shared memory policy in BCEdge can significantly reduce memory usage, thus achieving an optimal trade-off between throughput and latency.

We next illustrate that how BCEdge executes inference on six DNN models for a duration of 3,000 seconds in terms of throughput and latency, respectively. Fig. 10 shows the stacked graph of the accumulated throughput of each DNN model, and Fig. 11 reports the end-to-end latency of each DNN model over

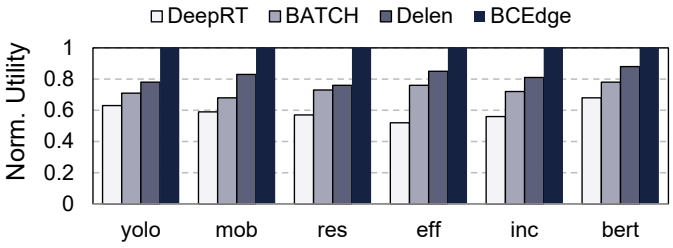


Fig. 9. Comparison of the normalized utility with six DNN models.

time. We can see that both the throughput and latency increase asymptotically between 0 and 1,500 seconds, which indicates that BCEdge is continuously optimizing our proposed utility function to identify the appropriate batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models for each DNN model. From 1,500 seconds on, both the throughput and latency are saturated, indicating that BCEdge has successfully fond the optimal joint actions in three dimensions search space within the constraint of memory and SLO. In addition, we note that BCEdge tends to sacrifice higher throughput for lower latency to achieve better utility.

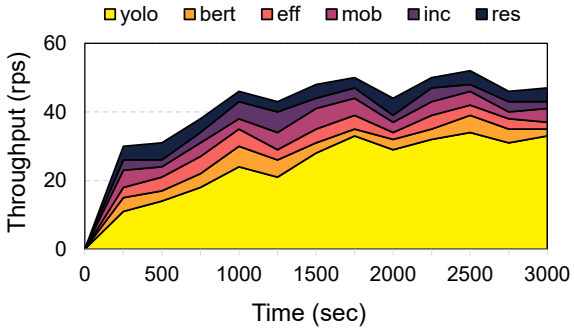


Fig. 10. Comparison of throughput with six DNN models. The scheduling duration of each DNN model for 3,000 seconds.

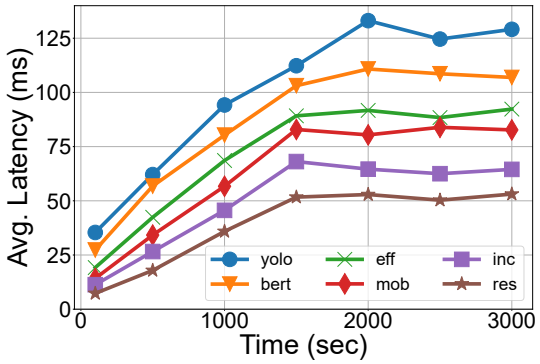


Fig. 11. Comparison of latency with six DNN models. The scheduling duration of each DNN model for 3,000 seconds.

C. Evaluation of Scalability

We additionally evaluate other four edge devices to evaluate the scalability of BCEdge. Table V provides the specific parameters of these heterogeneous edge devices compared with NVIDIA Xavier NX. We evaluate the scalability of

TABLE V
PERFORMANCE PARAMETERS OF EDGE DEVICES

Edge Devices	AI Performance	DRAM	CPU	GPU
NVIDIA Jetson Nano	0.47TFLOPS (FP16)	4GB 64-bit LPDDR4 25.6GB/s	4×Cortex-A57@1.4GHz	128×Maxwell@0.9GHz
NVIDIA Jetson TX2	1.33TFLOPS (FP16)	8GB 128-bit LPDDR4 59.7GB/s	6×Cortex-A57@1.4GHz	256×Pascal@1.3GHz
NVIDIA Jetson Xavier NX	21TOPS (INT8)	8GB 128-bit LPDDR4x 59.7GB/s	6×Carmel@1.4GHz	384×Volta@1.1GHz
NVIDIA Jetson Orin NX	100TOPS (INT8)	16GB 128-bit LPDDR5 102.4GB/s	8×Cortex-A78AE@1.4GHz	1024×Ampere@0.9GHz
NVIDIA Jetson AGX Orin	275TOPS (INT8)	64GB 256-bit LPDDR5 204.8GB/s	12×Cortex-A78AE@2.2GHz	2048×Ampere@1.3GHz

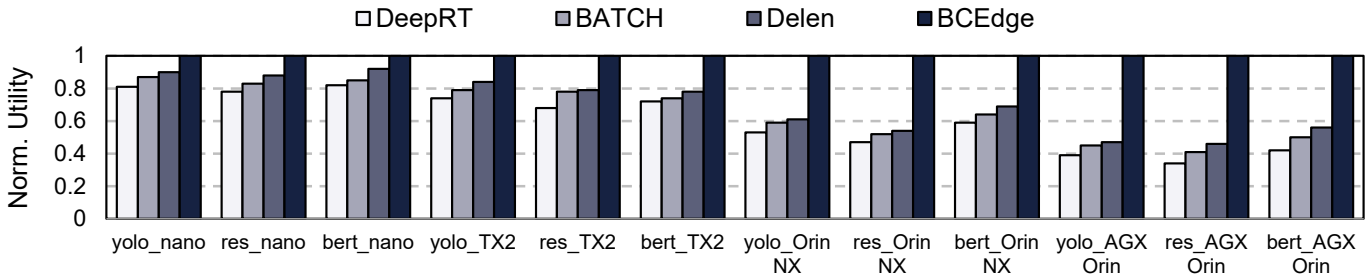


Fig. 12. The utility of heterogeneous edge devices. The more computility of edge devices, the higher the utility.

BCEdge using object detection (YOLO-v5), image classification (ResNet-18) and speech recognition (TinyBERT) DNN models, respectively. Fig. 12 reports the utility of BCEdge on four heterogeneous edge devices compared with baselines. We can see that BCEdge outperforms the baselines on all heterogeneous edge devices. Due to the image classification has the least computing resources, the ResNet-18 DNN model, therefore, has more optional batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models to better trade off throughput and latency than YOLO-v5 and TinyBERT. Similar results are also seen in Fig. 9. Even for Jetson nano with the weakest computility, the utility of BCEdge can be improved by 20% 15%, and 10% on average compared with DeepRT, BATCH and Ďelen, respectively. Since Jetson AGX Orin has the highest computility to configure more batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models. Thus, BCEdge receives the highest performance improvement, and its average utility is 62%, 55% and 50% higher than DeepRT, BATCH and Ďelen, respectively.

Fig. 13 reports the throughput and latency of four heterogeneous edge devices corresponding to Fig. 12. As shown in Fig. 13, BCEdge also has a significant performance improvement on the DNN models with fewer computing resources and the edge devices with higher computility. Even for Jetson Nano, BCEdge also achieves higher throughput and lower latency compared to all baselines. Overall, BCEdge exhibits flexible scalability that can adapt to heterogeneous resource-constrained edge devices.

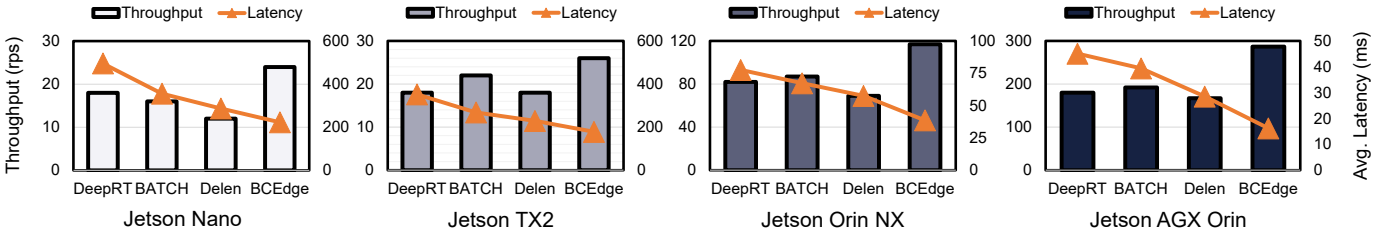
D. Evaluation of Interference Prediction Model

In this section, we evaluate the proposed interference prediction model in BCEdge with different requests per second (rps) on SLO violation rate. The interference prediction model records total 2000 inference interference data with one second period for each DNN model. Among the 2000 pieces of collected data, we randomly select 1600 pieces of execution data as training data and 400 pieces of data for validation. Fig. 14 presents the cumulative distribution of the prediction errors for our DNN-based interference model compared to a linear regression model [4], [52]. It can be seen that our proposed DNN-based model can predict up to 90% of cases with an error rate of 2.69%, and up to 95% of the cases if an error rate of 3.25% is allowed, and the error rate is reduced by half compared to the linear regression model. Since the model interference we observed in Fig. 2 is not a simple linear relationship, the linear regression model has a higher prediction error. In contrast, out proposed DNN-based interference model considers the memory utilization of edge devices and the collected historical data, which can accurately predict the interference latency.

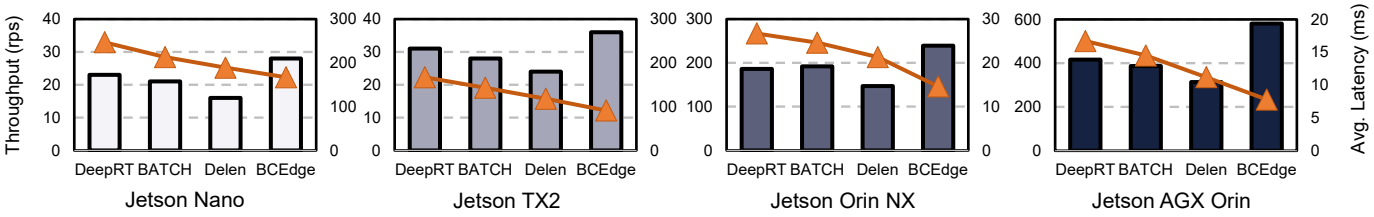
Fig. 15 shows the cumulative distribution of SLO violation rate at 30 rps for BCEdge with/without the interference prediction model. We analyze the SLO violation rate for a scheduling duration of 3000 seconds in Fig. 10. Our proposed model can reduce the SLO violation rate from 9.2% to 4.1% compared to BCEdge without the interference prediction model. It illustrates that the interference prediction model can improve the robustness of BCEdge and significantly reduce SLO violation rate.

We also evaluate the SLO violation rate by consistently increasing the requests per second (rps). As shown in Fig. 16, BCEdge has the lowest SLO violation rate compared to all baselines, which is 48%, 40% and 63% lower than DeepRT, BATCH and Ďelen on average, respectively, and the SLO violation rate of BCEdge is within 5% even at 40rps. Since the soft real-time scheduler in DeepRT is only suitable for DNN models without strict SLO budgets, it has a higher SLO violation rate than BCEdge. Despite BATCH leverages an optimizer to provide inference tail delay guarantees, it ignores the performance interference and memory contention among multiple DNN models, the SLO violation rate, therefore, is also higher than BCEdge, but lower than DeepRT. Delen does not consider the SLO budget for inference requests and thus has the highest SLO violation rate.

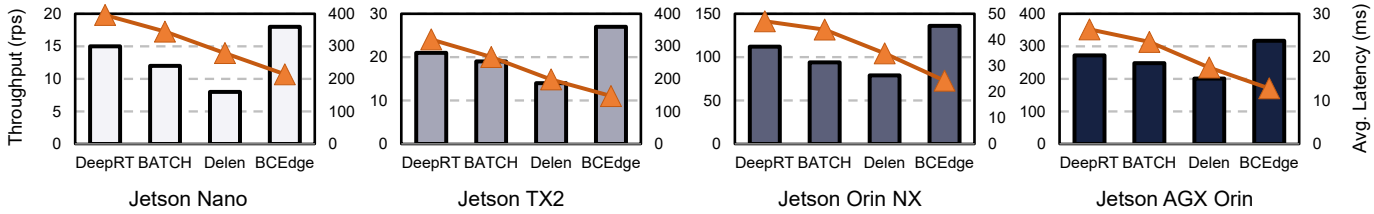
This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2024.3409701



(a) YOLO-v5 (yolo)



(b) ResNet-18 (res)



(c) TinyBERT (bert)

Fig. 13. The throughput and average latency of heterogeneous edge devices. The more computing resources of edge devices, the higher the throughput and the lower the average latency.

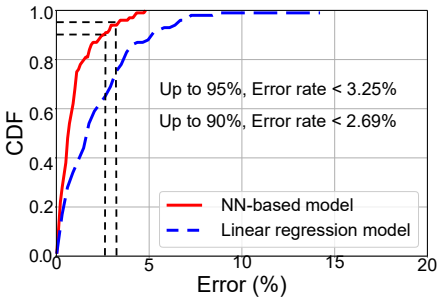


Fig. 14. Cumulative distribution of relative error rate. Our proposed DNN-based model can predict up-to 95% of cases with less than 3.25% error rate.

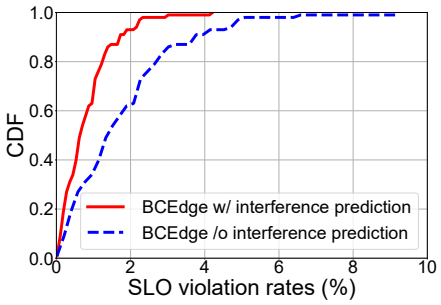


Fig. 15. Cumulative distribution of SLO violation rate with 30rps. Our proposed interference prediction model can achieve the SLO violation rate within 4%, compared to up to 9.2% SLO violation rate without the interference prediction model.

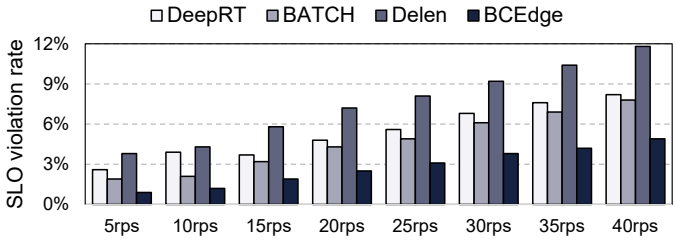


Fig. 16. Comparison of latency service level objective (SLO_L) violation rate with real-world six DNN model benchmarks under different rps. Benefit from the interference prediction model, BCEdge has the lowest SLO violation rate.

E. Overhead Analysis

Runtime Latency: in order to evaluate the runtime latency imposed by scheduler, we compare BCEdge with three baselines in terms of the average scheduling latency. Fig. 17 depicts these scheduling latency. As observed, BCEdge has a lower scheduling latency due to its scheduler leverages the branch-based DRL scheduling algorithm, which reduces the scheduling latency by executing actions in parallel. Specifically, the average scheduling latency of BCEdge is 26%, 43% and 35% lower than that of DeepRT, BATCH and Ďelen, respectively. It demonstrates that BCEdge can efficiently schedule batch and concurrent requests with extremely low scheduling latency. Note that we did not evaluate the overhead of performance profiler and interference prediction model as their overheads are negligible.

Memory Usage: in this section, we evaluate the memory usage of BCEdge for all DNN models compared to baselines. As shown in Fig. 18, the memory usage of BCEdge is 45%,

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2024.3409701

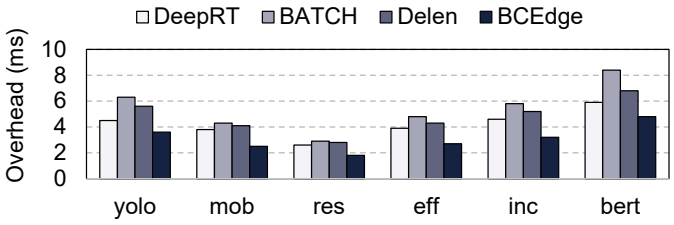


Fig. 17. Comparison of scheduling latency with six DNN models.

39% and 30% less than that of DeepRT, BATCH and Delen on average, respectively. Intuitively, the shared memory policy in BCEdge significantly reduces memory usage by sharing the parameter space and runtime space of multi-DNN model and multi-instance. In contrast, other baselines do not optimize the memory of inference service concurrency, which intensifies the memory usage among multiple DNN models and multi-instance and increases the risk of memory contention, thereby interfering with the trade-off between throughput and latency.

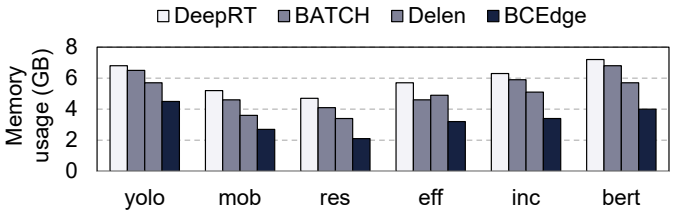


Fig. 18. Comparison of memory usage with six DNN models.

Scheduling Overhead: in this section, we discuss the scheduling time duration of BCEdge. As shown in Fig. 19, the scheduling time duration is negligible (less than 1%), compared to the time for each component of the end-to-end latency. This is due to our proposed branch-based reinforcement learning scheduling algorithm. It significantly reduces the scheduling time duration by replacing the sequential decisions in the traditional DRL algorithm with parallel decisions. In addition, we also discuss the possibility of combining heuristic-based solution with our proposed DRL-based solution in Section VII.

VII. Discussion

Request Arrival Rate Adaptation. Similar to the baseline approaches, BCEdge does not adapt the request arrival rate and this challenge can be left to future work. We plan to decouple the request arrival rate adaptation decision and scheduling algorithms [53], based on analyzing different request arrival patterns (such as Poisson distribution, normal distribution, random arrival, *etc.*) to better ensure QoS.

Scalability. We believe that BCEdge has the potential to be scaled beyond a single edge device and can support computing clusters of any size. However, due to network bandwidth fluctuations, additional communication latency is inevitably introduced. We plan to design a bandwidth adaptation policy to help with BCEdge scalability.

Integrated with DNN Compiler. The performance improvements of BCEdge benefit from model-level batching and

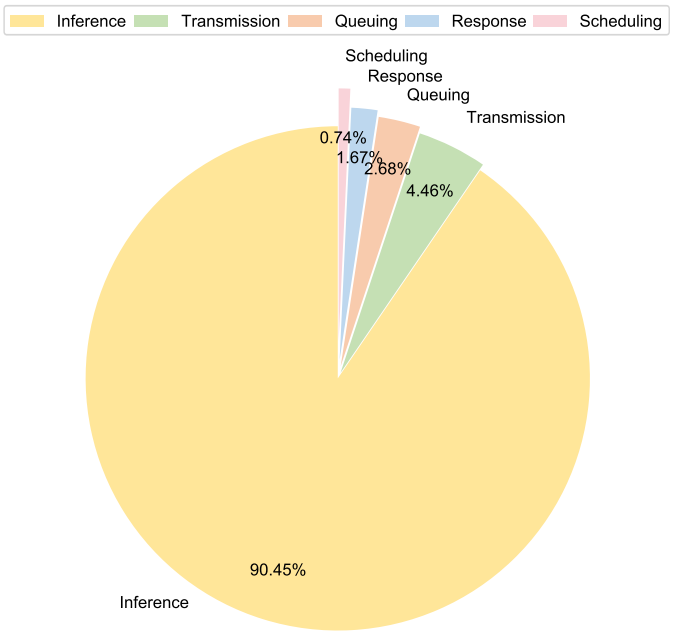


Fig. 19. Average scheduling time duration vs. Average time for each component of the end-to-end latency. We use NVIDIA Xavier NX edge devices to infer the six DNN models in Table IV.

concurrent scheduling, but still suffer from memory contention among multiple DNN models. To this end, we plan to integrate with DNN compilers to efficiently utilize the limited computational resources of edge devices. For instance, representative DNN compilers like TVM [28] can generate high-performance DNN operators with low latency using auto-tuning [54], based on fine-grained operator-level scheduling policies. BCEdge can serve as a post-compiling runtime to ensure that on-device resources are fully utilized during runtime in an adaptive manner.

Combine with Other Approaches. BCEdge can work symbiotically with other optimized DNN inference approaches, such as model compression [5], early exit [55] and edge-cloud collaborative inference [56]. In these ways, it becomes possible to achieve better throughput and latency tradeoffs, as well as higher resource utilization, enabling efficient execution of DNN model inference in resource-constrained edge computing environments. To optimize the scheduling time duration, a promising approach is to combine our proposed reinforcement learning-based scheduling solution with a heuristic-based scheduling solution. To be more specific, the heuristic-based scheduling solution is used for initial scheduling, and the reinforcement learning-based scheduling solution is used for runtime scheduling. We believe that this hybrid scheduling solution can better avoid SLO violation by reducing the scheduling time duration.

Hardware-Aware Latency Prediction. The basis for providing DNN inference services with high QoS is efficient and accurate inference prediction. Our proposed latency predictor currently depends on historical latency data and the memory utilization of edge devices. Yet, in practice, especially on edge devices with different architectures (such as CPU, GPU, NPU, and FPGA, *etc.*), it is hard to maintain stable prediction performance without having a detailed understanding of the

hardware's internals. In future work, we plan to explore a latency predictor for hardware-aware neural architecture search (NAS) [57] to accurately and efficiently predict DNN inference latency on different edge devices.

VIII. Conclusion

Providing multi-DNN model inference service with batching and concurrent execution on resource-constrained edge devices presents both promising opportunities and significant challenges, which involves guaranteeing SLO budgets for inference requests and reducing memory contention. In this paper, we propose BCEdge, an adaptive, SLO-aware scheduling framework for multiple DNN models inference service. BCEdge enables batching and concurrent inference for edge intelligent applications on edge devices to achieve both high-throughput and low-latency. The key to BCEdge is the learning-based scheduler with the parallel branches, which co-optimizes batch size, the number of concurrent instances for multiple DNN models and shared memory configuration among multiple DNN models automatically. Additionally, the shared memory policy and lightweight DNN-based prediction model in BCEdge aim to reduce memory usage and achieve SLO-aware, respectively. Extensive experiments show that BCEgde outperforms state-of-the-art schemes in improving the throughput-latency trade-off, reducing SLO violation rate and memory usage.

References

[1] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system." in *NSDI*, vol. 17, 2017, pp. 613–627.

[2] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[3] NVIDIA Corporation. (2023) Nvidia triton inference server. Accessed on 2023-12-18, Version 2.19.0. [Online]. Available: https://developer.nvidia.com/nvidia-triton-inference-server

[4] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 199–216.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[6] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.

[7] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 2019, pp. 3009–3018.

[8] D. Feng, C. Haase-Schütz, L. Rosenbaum, H. Hertlein, C. Glaeser, F. Timm, W. Wiesbeck, and K. Dietmayer, "Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 3, pp. 1341–1360, 2020.

[9] Y. Yang, H. Luo, H. Xu, and F. Wu, "Towards real-time traffic sign detection and classification," *IEEE Transactions on Intelligent transportation systems*, vol. 17, no. 7, pp. 2022–2031, 2015.

[10] N. Shvai, A. Hasnat, A. Meicler, and A. Nakib, "Accurate classification for automatic vehicle-type recognition based on ensemble classifiers," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 3, pp. 1288–1297, 2019.

[11] X. Ouyang, X. Shuai, J. Zhou, I. W. Shi, Z. Xie, G. Xing, and J. Huang, "Cosmo: contrastive fusion learning with small data for multimodal human activity recognition," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022, pp. 324–337.

[12] Ultralytics. (2023) Yolov5. Accessed on 2023-11-14, Version 7.0. [Online]. Available: https://github.com/ultralytics/yolov5/tree/v7.0

[13] NVIDIA Corporation. (2023) Nvidia tensorrt. Accessed on 2023-12-26, Version 8.2. [Online]. Available: https://developer.nvidia.com/tensorrt

[14] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo, "Veltair: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 388–401.

[15] Z. Yang, K. Nahrstedt, H. Guo, and Q. Zhou, "Deeprt: A soft real time scheduler for computer vision applications on the edge," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 271–284.

[16] Q. Liang, W. A. Hanafy, N. Bashir, A. Ali-Eldin, D. Irwin, and P. Shenoy, "Dělen: Enabling flexible and adaptive model-serving for multi-tenant edge ai," in *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation*, 2023, pp. 209–221.

[17] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[18] Y. Choi and M. Rhu, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 220–233.

[19] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "{DVABatch}: Diversity-aware {Multi-Entry}{Multi-Exit} batching for efficient processing of {DNN} services on {GPUs}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 183–198.

[20] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving." in *USENIX Annual Technical Conference*, 2019, pp. 1049–1062.

[21] W. Seo, S. Cha, Y. Kim, J. Huh, and J. Park, "Slo-aware inference scheduler for heterogeneous processors in edge platforms," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, pp. 1–26, 2021.

[22] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving." in *USENIX Annual Technical Conference*, 2021, pp. 397–411.

[23] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[24] X. Liu, B. Wu, X. Yuan, and X. Yi, "Leia: A lightweight cryptographic neural network inference system at the edge," *IEEE Trans. Inf. Forensics Secur.*, vol. 17, pp. 237–252, 2021.

[25] J. Hou, H. Liu, Y. Liu, Y. Wang, P.-J. Wan, and X.-Y. Li, "Model protection: Real-time privacy-preserving inference service for model privacy at the edge," *IEEE Trans. Dependable Secure Comput.*, 2021.

[26] Y. G. Kim and C.-J. Wu, "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1082–1096.

[27] W. Zhang, D. Yang, H. Peng, W. Wu, W. Quan, H. Zhang, and X. Shen, "Deep reinforcement learning based resource management for dnn inference in industrial iot," *IEEE Trans. Veh. Technol.*, vol. 70, no. 8, pp. 7605–7618, 2021.

[28] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[29] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 539–558.

[30] W. Cui, H. Zhao, Q. Chen, N. Zheng, J. Leng, J. Zhao, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[31] Z. Zhang, H. Li, Y. Zhao, C. Lin, and J. Liu, "Pos: An operator scheduling framework for multi-model inference on edge intelligent

computing," in *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, 2023, pp. 1–1.

[32] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th annual international conference on mobile computing and networking*, 2020, pp. 1–15.

[33] S. S. Mondal, N. Sheoran, and S. Mitra, "Scheduling of time-varying workloads using reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 9000–9008.

[34] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[35] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Trans. Mob. Comput.*, 2020.

[36] X. Wang, Z. Ning, and S. Guo, "Multi-agent imitation learning for pervasive edge computing: A decentralized computation offloading algorithm," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 411–425, 2020.

[37] J. Zou, T. Hao, C. Yu, and H. Jin, "A3c-do: A regional resource scheduling framework based on deep reinforcement learning in edge scenario," *IEEE Trans. Comput.*, vol. 70, no. 2, pp. 228–239, 2020.

[38] D. Shi, H. Gao, L. Wang, M. Pan, Z. Han, and H. V. Poor, "Mean field game guided deep reinforcement learning for task placement in cooperative multiaccess edge computing," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9330–9340, 2020.

[39] S. Tuli, G. Casale, and N. R. Jennings, "Mcds: Ai augmented workflow scheduling in mobile edge cloud computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2794–2807, 2021.

[40] J. Meng, H. Tan, X.-Y. Li, Z. Han, and B. Li, "Online deadline-aware task dispatching and scheduling in edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1270–1286, 2019.

[41] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[43] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 32, no. 1, 2018.

[44] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[45] Z. Zhao, N. Ling, N. Guan, and G. Xing, "Miriam: Exploiting elastic kernels for real-time multi-dnn inference on edge gpu," *arXiv preprint arXiv:2307.04339*, 2023.

[46] J. Jeong, S. Baek, and J. Ahn, "Fast and efficient model serving using multi-gpus with direct-host-access," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 249–265.

[47] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

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

[49] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[50] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[51] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "TinyBERT: Distilling BERT for natural language understanding," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Nov. 2020, pp. 4163–4174.

[52] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[53] V. Nigade, P. Bauszat, H. Bal, and L. Wang, "Jellyfish: timely inference serving for dynamic edge networks," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 277–290.

[54] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[55] Z. Zhang, Y. Zhao, and J. Liu, "Octopus: Slo-aware progressive inference serving via deep reinforcement learning in multi-tenant edge cluster," in *International Conference on Service-Oriented Computing*. Springer, 2023, pp. 242–258.

[56] Z. Zhang, Y. Zhao, H. Li, C. Lin, and J. Liu, "Dvfo: Learning-based dvfs for energy-efficient edge-cloud collaborative inference," *IEEE Transactions on Mobile Computing*, 2024.

[57] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7675–7688.

Ziyang Zhang Ziyang Zhang received the MS degrees in the School of Electronic Information and Optical Engineering, Nankai University, Tianjin, China, in 2020. He is currently working toward the PhD degree in the School of Computer Science and Technology, Harbin Institute of Technology (HIT), Harbin, China. His research interests include edge computing, machine learning system, and deep learning.

Yang Zhao Yang Zhao received the BS degree (2003) in electrical engineering from Shandong University, the MS degree (2006) in electrical engineering from the Beijing University of Aeronautics and Astronautics, and the PhD degree (2012) in electrical and computer engineering from the University of Utah. He was a lead research engineer at GE Global Research between 2013 and 2021. Since 2021, he has been at Harbin Institute of Technology, Shenzhen, where he is a research professor in the International Research Institute for Artificial Intelligence. His research interests include wireless sensing, edge computing and cyber physical systems. He is a senior member of the IEEE.

Huan Li Dr. Huan Li obtained her PhD degree in Computer Science from the University of Massachusetts at Amherst, USA in 2006. Her current research interests include AIoT, Edge intelligence, distributed real-time systems, and data science. She has served as program committee member for numerous international conferences including IEEE RTAS, ICDCS, RTCSA, etc. She is now a senior member of IEEE.

Jie Liu Jie Liu is a Chair Professor at Harbin Institute of Technology Shenzhen (HIT Shenzhen), China and the Dean of its AI Research Institute. Before joining HIT, he spent 18 years at Xerox PARC and Microsoft. He was a Principal Research Manager at Microsoft Research, Redmond and a partner of the company. His research interests are Cyber-Physical Systems, AI for IoT, and energy efficient computing. He received IEEE TCCPS Distinguished Leadership Award and 6 Best Paper Awards from top conferences. He is an IEEE Fellow and an ACM Distinguished Scientist, and founding Chair of ACM SIGBED China.