OINC: Efficient Online DNN Inference with Continuous Learning in Edge Computing

***Abstract*—Compressed edge DNN models usually experience decreasing model accuracy when performing inference due to data drift. To maintain the inference accuracy, retraining models with continuous learning is usually employed in the edge. However, online edge DNN inference with continuous learning faces new challenges. First, introducing retraining jobs leads to resource competition with the existing edge inference tasks, which will affect the inference latency. Second, retraining jobs and inference tasks exhibit significant differences in workload and latency requirements. These two jobs cannot adopt the same scheduling policy. To overcome the challenges, we propose an Online scheduling algorithm for INference with Continuous learning (`OINC`). `OINC` balances the latency of inference tasks and the completion time of retraining jobs with limited edge resources while ensuring the satisfaction of the inference task's service level objective (SLO) and meeting the deadlines of retraining jobs. `OINC` first reserves a portion of resources to complete all current inference tasks and allocates the remaining resources to retraining jobs. Subsequently, based on the reserved resource ratio, `OINC` invokes two sub-algorithms to select edges and allocate resources for each inference task and retraining job respectively. Compared with six state-of-the-art algorithms, `OINC` can reduce the weighted average latency by up to 23.7%, and increase the success rate by up to 35.6%.**

I. Introduction

An increasing number of artificial intelligence (AI) applications are being designed to run on devices, addressing various problems such as object recognition, augmented reality, autonomous driving, and more. AI applications often require significant computing and storage resources [1]. However, due to limitations in computing capacity, storage space, and battery capacity, a single device is insufficient for handling such tasks [2]. Meanwhile, traditional cloud-based approaches are constrained by high transmission latency, bandwidth costs, and the risk of privacy breaches [3]. Edge inference offloads the inference tasks from devices to edge servers, enhancing the processing capabilities of devices while meeting the latency requirements of tasks.

Edge computing offers limited resources, necessitating the deployment of compressed deep neural network (DNN) models [4]. However, compressed models have fewer weight parameters and shallower model structures, and struggle to adapt to significant data variations. Because of the deviation of input data from the training data, which is known as *data drift* [5], compressed models experience decreasing model accuracy when deployed in the edges [6]. For instance, in object recognition tasks, variations in object pose, scene density, and lighting over time can challenge edge DNNs in accurately identifying the objects of interest. To address the challenges posed by rapidly changing environments, *continuous learning*, also known as model retraining, has been proposed. Continuous learning collects real-time drift data and continuously retrains a personalized compressed model based on the original model for devices [6]. This approach enables edge DNN models to maintain high accuracy when processing real-world data streams, effectively adapting to the dynamic environment.

However, online edge DNN inference with continuous learning poses unique challenges. **First**, offloading retraining jobs on edges will result in resource competition with existing inference tasks. Allocating more resources to retraining jobs can expedite model retraining while resulting in increased inference latency because of fewer resources available for inference tasks. On the other hand, the prompt completion of model retraining facilitates timely improvements in inference accuracy. Balancing the allocation of resources becomes more challenging due to the interdependence between inference tasks and retraining jobs. **Second**, there is a substantial difference in the computing workload between retraining and inference tasks. Given that the workload of inference tasks is significantly smaller than retraining jobs [7], [8], it is essential to properly distinguish them when scheduling. Otherwise, inference tasks may be allocated minimal resources, leading to excessive inference latency. Similarly, the time scale for these tasks differs as well, with inference tasks typically measured in milliseconds and retraining jobs measured in minutes or hours. There is a substantial difference in the impact of inference latency and retraining completion time on minimizing the total latency and completion time. **Last but not least**, in practical, both retraining and inference tasks arrive online. While the arrival rate of some inference tasks can be predicted based on historical data, the generation of retraining tasks depends entirely on the data drift, making it unpredictable. Therefore, it is difficult to design efficient online scheduling algorithms to quickly response the dynamic changes while satisfying the inference task's service level objective (SLO) and meeting the deadlines of retraining jobs.

Existing papers on continuous learning focus primarily on how to perform retraining and when to update models on edges to minimize costs and meet requirements for latency and accuracy [7], [9]–[12]. They process retraining jobs on the cloud or a fixed edge, resulting in excessive costs. Their approaches hinder the timely completion of jobs, particularly in the multi-task scenario. Existing schedulers in edges only consider either inference tasks or retraining jobs individually [13]–[16], but the differences between these two types of jobs make it difficult to schedule them together. Some approaches use heuristic algorithms [17] or iterative algorithms [18], but

these methods often require long scheduling times and struggle to meet the latency requirements of inference tasks. One recent work [19] addresses the resource allocation problem for inference tasks and retraining jobs on edge servers by using a micro-profiler to estimate the accuracy and resource requirements of different configurations for model retraining. However, it does not consider the online arrival of inference and retraining tasks. We will discuss in details in Sec. II.

To address the aforementioned challenges, to the best of our knowledge, we are the **first** to propose an Online scheduling algorithm for INference with Continuous learning (OINC) to simultaneously schedule multi-inference tasks and retraining jobs in edge networks. Our approach aims to minimize the weighted sum of the latency of inference tasks and the completion time of retraining jobs while ensuring the inference SLO and the deadline for retraining jobs.

To handle the differences between inference and retraining, we propose a reservation algorithm that allocates resource for both inference and retraining based on their respective workloads. We devise distinct online scheduling algorithms for inference tasks and retraining jobs respectively, employing spatial and temporal resource sharing strategies. In the inference scheduling stage, we leverage a reinforcement learning (RL) algorithm based on soft actor-critic with discrete actions (SAC-D) to learn the coupling relationships between two discrete decisions (i.e., task offloading and resource allocation). By combining an actor-critic architecture with discrete actions, the RL algorithm facilitates efficient adaptation to the dynamic generation patterns of inference tasks. This approach enables quick decision-making, thereby meeting the latency requirements of inference tasks that are sensitive to response time. In the retraining scheduling stage, retraining jobs are characterized by their large computational requirements, unpredictable duration, and high switching costs, making them unsuitable for frequent adjustments of resource space allocation. Therefore, we employ a resource time-sharing allocation strategy. We select a retraining job to process in each edge based on factors such as job weight, computing workload, and completion deadline at each time slot. This approach reduces preemption and job switching while ensuring the timely handling of urgent tasks. When the reserved resources for inference are insufficient, the OINC algorithm can prioritize the completion of inference tasks by temporarily pausing retraining jobs and allocating the available computing resources to inference tasks. We summarize our main contributions as follows:

- *Joint Online Scheduling of Inference and Retraining Model*. We analyze the characteristics of inference tasks and retraining jobs, and explicitly model the inference latency and retraining completion time in multi-task edge scenario. Based on that, we formulate an online weighted latency and completion time minimization problem.
- *Online Scheduling Algorithm OINC*. To solve the formulated problem, OINC first reserves part of its resources for inference tasks, and then decomposes the problem into two subproblems, inference scheduling and retraining scheduling. For the first subproblem, OINC utilizes the RL training algorithm based on SAC-D to determine task offloading and resource allocation efficiently. For the second subproblem, the retraining scheduling algorithm dispatches each job to an edge and determines which retraining job will be processed on its edge at each time slot. It can be proved that the retraining scheduling algorithm is $O(\frac{1}{\epsilon})$-competitive with $(1+\epsilon)$-speed augmentation, where $\epsilon \in (0,1)$.
- We evaluate the effectiveness of OINC through extensive experiments. The results show that: i) OINC achieves both low inference latency and low retraining completion time, compared to six baselines; ii) OINC reduces the weighted average latency by up to 23.7%. OINC improves the success rate by up to 35.6%.

II. Related Work

A. Inference Scheduling in Edge Computing

Existing research on edge inference mainly focused on inference latency minimization [8], [20]–[22], inference cost minimization [23], and energy utility minimization [17], [24], etc. Liu *et al.* [8] proposed an approach for edge-cloud orchestrated computing to minimize the latency of tasks. Chu *et al.* [17] jointly optimized service selection, resource allocation, and task offloading to maximize users' QoE. Fan *et al.* [20] proposed an iterative algorithm based on Lyapunov optimization to minimize the total latency of tasks while ensuring task queue stability. Liu *et al.* [21] proposed a method based on deep Q-networks to solve the problem of federated edge service placement and computing resource allocation. Liu *et al.* [22] additionally considered the heterogeneity of edge resources and proposed a jointly determined algorithm to minimize the total latency. Eshraghi *et al.* [23] formulated the problem as a mixed-integer program and designed an algorithm to minimize the average cost. Jiang *et al.* [24] focused on task latency and energy consumption, and used Lyapunov optimization to solve the joint offloading and allocation problem. Moro *et al.* [25] employed convex programming to maximize fairness and diverse requirements of different inference services. However, due to the differences between inference tasks and retraining tasks, especially in terms of workload and processing time, existing inference scheduling approaches cannot jointly handle both tasks.

B. Continuous Learning and Model Updating

A common approach to update retraining models is using new data [9]–[11]. Chen *et al.* [9] addressed the requirement to perform model retraining on the cloud server and subsequently update the model to edges while minimizing the data transfer volume for bandwidth limits. Zhang *et al.* [10] designed an active learner to sample drift data for labeling and use labeled data for continuous learning, which can reduce costs while maintaining the accuracy of edge models. The timing of model updates is also a hot topic. Tian *et al.* [7] provided two update policies, best-effort, and cost-aware, to decide when to update models to cope with dynamic data, with and without considering training cost, respectively. Aleksandrova

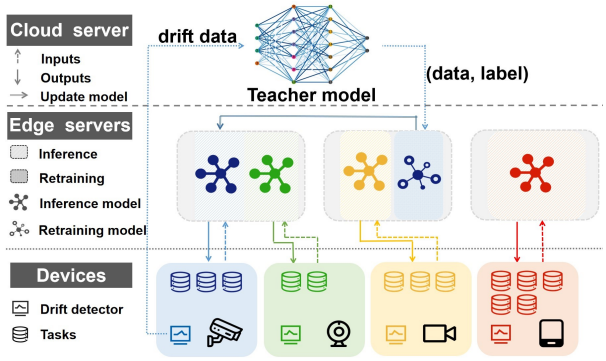

Fig. 1: System Overview.

et al. [12] proposed a method to determine update scheduling and minimize the network overhead based on the Optimal Stopping Theory (OST) principles. These researchers designed the system for continuous learning and studied the approach to update models. However, they have not considered the scheduling problem of retraining jobs, not to mention jointly scheduling with inference tasks.

C. Retraining Scheduling in Edges

The previous ML scheduling methods [13]–[16] are not suitable for scheduling retraining jobs and inference tasks together. Bhardwaj *et al.* [19] jointly allocated resources for inference and retraining tasks to maximize the average accuracy. However, their offline algorithm is not suitable for the online multi-task edge environment. Bhattacharjee *et al.* [26] presented a scheduler that reduces the total retraining time while considering heterogeneity among the edges and the resource interference caused by the colocation of the model updating jobs and latency-critical tasks. But [26] only considered the inference tasks as a known background program, and did not consider the resource allocation for both inference tasks and retraining jobs. Therefore, an approach to handle the joint online scheduling problem needs to be devised.

III. System Model

A. System Overview

As shown in Fig. 1, we consider a three-layer device-edge-cloud system consisting of a cloud, multiple edges, and multiple devices. Devices are equipped with radio access networks to communicate with edges, while the cloud is connected to the devices and edges through a wide area network. We assume that the cloud has sufficient storage resources and computing resources, and can assist the system in performing high-precision inference. The edges, with different computing capabilities (i.e., maximal CPU frequency) and wireless bandwidth, store pre-trained compressed models for inference. The devices receive input data online and generate inference tasks. We associate each device with a specific compressed DNN model, which is used to process the inference tasks from that device. Due to the insufficient computing resources on devices and the high cost of data transmission to the cloud, devices rely on edges to efficiently process inference tasks. Let $[X]$ denote the set $\{1, 2, ..., X\}$. The set of devices is denoted as $[I]$, and the set of edges is denoted as $[J]$. To facilitate the study of the online system, we discretize time into equal-interval time slots, denoted by $[T]$. Each time slot of $t \in [T]$ represents a decision interval that matches the change of the system dynamics. In practice, the length of a time slot (i.e., 1 second) is longer than a typical end-to-end latency of DNN model inference.

Inference Task Information. In this work, we consider the online stochastic generation of heterogeneous inference tasks from devices. Each device will continuously generate or receive input data samples. At each time slot, devices offload inference tasks to edges for completion. We assume that tasks generated by each device $i \in [I]$ at each time slot $t \in [T]$ have different input data and workload, denoted as $Task_i^t = \{d_i^t, c_i^t, SLO_i^t\}$. Here, d_i^t represents the input data size of the task i (i.e., the amount of data to be delivered to the edge in MB), c_i^t indicates the computing workload of the task i (i.e., the number of CPU cycles required in total to complete the task), and SLO_i^t represents the service-level objective requirement (in terms of latency) of the task. The SLOs for inference tasks may vary, but they all require completion within a single time slot. Note that each device only knows which task is generated in the current time slot, without any information in the future.

Model Retraining. In dynamic environments, the data generated continuously and sequentially in real time is referred to as data streams [27]. In our system, each device has its own data stream, and the distribution of these streams may vary over time. The *concept drift* is used to represent the change of data streams in data characteristics. The DNN models, which are hosted in edges to process inference tasks, are assumed to be compressed models due to the limited storage resources of the edges. When confronted with concept drift, the precision of compressed models tends to deteriorate [6], leading to a compromised quality of inference. Consequently, it becomes imperative to retrain the models.

To detect the concept drift, we utilize an unsupervised *drift detector* IBDD [28] for each device. The drift detectors can identify the sample with concept drift in data streams without depending on labeled data or the outputs of DNNs. Since manual labeling is not feasible for the continuous retraining system, we deploy a *teacher model* in the cloud to label the drift data used for retraining. The teacher model is a highly accurate but computationally expensive model with a deeper architecture and a large number of weights. The system transfers the drift data samples from devices to the cloud, and obtains the ground-truth labels of them from the teacher model. Given that the communication between devices and the cloud is costly and time consuming, the teacher model is used only to label drift data samples, which are a small fraction of the data streams of devices.

The inference results from edges and the concept drift data from devices will both be transmitted to the cloud. The cloud will compare the result with the ground-truth labels to calculate the accuracy of edge DNNs. When the inference accuracy of the edge DNN model is below a predefined threshold, the cloud will generate a retraining job. Prior

research on continuous learning suggests that retraining a model solely on drift samples can cause the model to forget its original knowledge [29]. To address this issue, we propose a retraining approach that randomly replaces a portion of the original training samples with the current drift data during each retraining epoch. The original training samples are stored in each edge, and the drift data samples (with labels) are transferred from the cloud to the selected edge. This approach retains a part of the historical data and constructs a retraining set with a fixed total number of samples.

Let $[K]$ denote the set of retraining jobs. The retraining job k is denoted as $Job_k = \{i_k, a_k, w_k, D_k\}$, where i is the index of the model, a_k denotes the generated time, w_k represents the workload of job (i.e., the number of CPU cycles required in total to retrain the model i), D_k is the deadline to complete the job.

System Workflow. At time slot t, the system works as follows, shown in Fig. 1:

- *Step 1: Detection.* Devices receive input data samples, package them into an inference task at each time slot, and offload them to an edge. Each device has a drift detector. If a drift data sample is found, the device will also transmit it to the cloud.
- *Step 2: Labeling.* The drift data samples are labeled by the teacher model in the cloud. The cloud compares whether the ground-truth labels from the teacher model are consistent with the inference results from the edge models, and calculates the accuracy of each edge model. When the accuracy is lower than the predefined threshold, the cloud will inform the edge to start a retraining job. The drift data samples and labels will be transmitted to the edge for retraining.
- *Step 3: Inference and Retraining.* Both the inference tasks and the retraining jobs are processed in edges. Given the limited computing capacity of the edge, the computation resources must be carefully allocated among the inference and retraining requests, to minimize both the latency of all inference tasks and the completion time of retraining jobs. After a model is retrained, the new parameters of the model will be updated to each edge to continue inference.

Decision Variables. After the generation of inference tasks and retraining jobs at time slot t, the decisions made include: i) $y_{i,j}^t \in \{0,1\}$, whether edge j is selected for inference task i at t; ii) $z_{i,j}^t \in [0,1]$, the proportion of computing resources allocated to inference task i in edge j at t; iii) $v_{i,j}^t \in [0,1]$, the proportion of bandwidth allocated to device i in edge j at t. iv) $x_{k,j}^t \in \{0,1\}$, whether edge j is selected for retraining job k at t; v) $u_{k,j}^t \in [0,1]$, the proportion of computing resources allocated to retraining job k in edge j at t.

B. Inference Latency

In what follows, we analyze the inference latency, which includes the time taken for uploading input data, the computation time, and the time taken for transmitting the output data back. Similar to some other works [30], we ignore the output data of tasks due to their small size. Therefore, the downlink transmission time for sending output back to devices is omitted in our work.

Transmission Latency. The device communicates with the edge via a wireless network connection. The transmission interference can be ignored by exploiting the orthogonal frequency division multiple access [31]. Then, the data transmission rate between device i and edge j at time slot t is obtained by the Shannon formula as $r_{i,j}^t = v_{i,j}^t B_j * \log_2(1+\frac{\rho_i g_{i,j}^t}{\sigma^2})$, where $v_{i,j}^t$ denotes the proportion of bandwidth allocated to model i in edge j at time slot t, B_j indicates the wireless channel bandwidth of edge j, ρ_i indicates the transmission power of device i, $g_{i,j}^t$ represents the channel gain between device i and edge j, and σ denotes the power of the Gaussian noise in the device-to-edge channel. The input data transmission latency between device i and edge j can be calculated by $\frac{d_i^t}{r_{i,j}^t}$.

Computation Latency. The computation latency is the execution time of the DNN model in the edge, which can be represented by $\frac{c_i^t}{z_{i,j}^t f_j}$, where $z_{i,j}^t$ stands for the computing resource allocated to inference task i in edge j, and f_j denotes the computing capacity of edge j.

Therefore, the total inference latency of task from device i in edge j at time slot t is calculated as: $\tau_{i,j}^t = \frac{d_i^t}{r_{i,j}^t} + \frac{c_i^t}{z_{i,j}^t f_j}$.

C. Completion Time for Retraining

Due to the limited resources in edges, a retraining job may not be processed immediately after being assigned to an edge, and the processing may be interrupted by more urgent retraining jobs or inference tasks. Therefore, our goal is to minimize the completion time of retraining jobs, rather than the processing latency, to avoid excessive waiting time. At time slot b_k, denoting as the completion time of job k, all the workload of the job has been fully processed and finished, i.e., $b_k = \text{argmin}_{t>a_k}(\sum_j x_{k,j}^t u_{k,j}^t f_j \leqslant 0)$.

D. Problem Formulation

The objective of the system is to minimize DNN inference latency and retraining job completion time, subject to the SLOs of inference tasks, the deadline of retraining jobs, and the limited resources of edges. We use α_t to denote the weights for inference tasks generated at time t. β_k is used to denote the weights for retraining job k.

The online problem for DNN inference and retraining can be formulated as follows:

$$\text{minimize} \quad P = \sum_t \sum_i \sum_j \alpha_t y_{i,j}^t \tau_{i,j}^t + \sum_k \beta_k b_k, \tag{1}$$

$$\text{s.t.} \quad \sum_j y_{i,j}^t \tau_{i,j}^t \leqslant SLO_i^t, \ \ \forall i, \forall t, \tag{1a}$$

$$\sum_j y_{i,j}^t = 1, \ \ \forall i, \forall t, \tag{1b}$$

$$\sum_i v_{i,j}^t \leqslant 1, \ \ \forall j, \forall t, \tag{1c}$$

$$\sum_j x_{k,j}^t = 1, \ \ \forall k, \forall t, \tag{1d}$$

$$\sum_{t=a_k}^{b_k} \sum_j x_{k,j}^t u_{k,j}^t f_j \geqslant w_k, \ \ \forall k, \tag{1e}$$

$$\sum_i z_{i,j}^t + \sum_k u_{k,j}^t \leqslant 1, \ \ \forall j, \forall t, \tag{1f}$$

$$x_{k,j}^t \in \{0,1\}, u_{k,j}^t \in [0,1], \ \ \forall k, \forall j, \forall t, \tag{1g}$$

$$y_{i,j}^t \in \{0,1\}, z_{i,j}^t \in [0,1], v_{i,j}^t \in [0,1], \ \ \forall i, \forall j, \forall t \tag{1h}$$

Constraint (1a) means that the total inference latency of the task cannot exceed its SLO. Constraint (1b) guarantees that only one edge is selected for each generated inference task. Constraints (1c) and (1f) ensure that the edge bandwidth and computing resources allocated to retraining jobs and inference tasks do not exceed the resource capacity. Constraints (1d) means that each retraining job only can select one edge to complete its process. Constraint (1e) guarantees that each job is allocated sufficient resources to retrain. [1]

Challenge. First, our aim is to address the above problem (1) in an online manner. Second, the problem is a mix-integer non-linear optimization problem, which is NP-hard. It is different to solve the problem in the offline setting and will be more challenging to solve in an online setting. Finally, the interdependence among the decision variables of the problem (1) introduces additional complexity, making the problem more difficult to address. For instance, in constraint (1f), the resource allocation between inference and retraining mutually influences each other.

IV. The Design of OINC

A. Main Idea

In this section, we propose our online algorithm OINC, followed by the theoretical analysis. OINC comprises the following stages:

i. When a retraining job is generated, OINC first checks whether there are any edges that have not been utilized for retraining jobs. If so, OINC invokes the reservation algorithm (Alg. 2 in Sec. IV-B). It reserves a portion of computing resources for inference tasks and calculates the proportion of resources that can be used for retraining. Otherwise, OINC proceeds to the next stage.

ii. OINC invokes the inference scheduling algorithm (A_{inf} in Alg. 4) to process inference tasks and allocate resources to tasks from reserved resources. A_{inf} is based on an actor network, which is trained by the RL training algorithm (Alg. 3).

iii. For retraining jobs, OINC employs the retraining scheduling algorithm (A_{ret} in Alg. 5) to schedule them based on the reserved proportion of resources. A_{ret} calculates the weighted density for each job, and then dispatches the job based on it. Each edge sorts the assigned retraining jobs and decides which job to process at each time slot.

Design of OINC. Our online algorithm framework OINC is presented in Alg. 1. At each time slot, OINC receives the information of retraining jobs $Job_k = \{i, t, w_i^t, D_i^t\}$ and inference tasks $Task_i^t = \{d_i^t, c_i^t, SLO_i^t\}$ (lines 3-4). According to the information, OINC updates the number of retraining jobs num_{job} and the number of edges without retaining jobs num_{edge} (line 5). Next, OINC invokes the reservation algorithm (Alg. 2) to allocate the resource for retraining in each edge p_j^r when there are edges that have not been deployed with retraining jobs (lines 7-9). p_j^r represents the proportion of resources available for retraining jobs in edge j, and then the total proportion of resources allocated to inference tasks in edge j is $1 - p_j^r$. Finally, OINC invokes the inference scheduling algorithm (Alg. 4) and calls the retraining scheduling algorithm (Alg. 5) with p_j^r as input (lines 11-12).

[1]We unified the ranges of the inference latency and retraining completion time values using the min-max normalization approach based on their actual values obtained from pertaining. The weights assigned to inference tasks and retraining jobs are used to indicate their respective priorities.

Algorithm 1 OINC Algorithm

Input: $I, J, T, B_j, f_j, [Jobs]_t, [Tasks]_t, \forall t, \forall j$
Output: $x_{k,j}^t, u_{k,j}^t, y_{i,j}^t, z_{i,j}^t, v_{i,j}^t, \forall t, \forall i, \forall j, \forall k$
1: Initialize $num_{job} = 0, num_{edge} = J, p_j^r = 0, Q_j = \varnothing, \forall j$;
2: **for** $t = 1$ to T **do**
3: Receive information of retraining jobs $[Jobs]_t$;
4: Receive information of inference tasks $[Tasks]_t$;
5: Update num_{job}, num_{edge} based on $[Jobs]_t$ and $p_j^r, j \in [J]$;
6: **if** $num_{job} \neq 0$ **then**
7: **if** $num_{edge} \neq 0$ **then**
8: With t, num_{job}, num_{edge} as input, invoke the reservation algorithm (Alg. 2) to get $p_j^r, \forall j$;
9: **end if**
10: **end if**
11: With $p_j^r, \forall j$ as input, invoke A_{inf} (Alg. 4) to schedule inference tasks;
12: With $p_j^r, \forall j$ as input, invoke A_{ret} (Alg. 5) to schedule retraining jobs;
13: **end for**

Algorithm 2 Reservation Algorithm

Input: t, num_{job}, num_{edge}
Output: $p_j^r, \forall j \in [J]$
1: Obtain res_{min} with the historical and current workload;
2: Calculate $num = \min(num_{job}, num_{edge})$;
3: Select the num edges with maximum computing capacity into $[J']$;
4: **for** $\alpha/\beta \in [R]$ **do**
5: Calculate $p_{j'}^r, j' \in [J']$ with Eq. (2);
6: Calculate the latency of inference tasks and retraining jobs $l_{\alpha/\beta}$ at time t;
7: Store $(l_{\alpha/\beta}, \alpha, \beta)$ into $[L_t]$;
8: **end for**
9: Select α_t, β_k in $[L_t]$ with minimize $l_{\alpha/\beta}$;
10: Calculate $p_{j'}^r, j' \in [J']$ with (2) based on given α_t, β_k;
11: **return** $p_j^r, \forall j \in [J]$;

B. Resource Reservation

The reservation algorithm partitions resources for retraining jobs while reserving a portion of resources for inference tasks. The workflow of the reservation algorithm is as follows:

Firstly, we obtain the minimum computing resource required by inference tasks res_{min} based on the historical input patterns and the current workload of tasks (line 1). num is set as the minimum of num_{job} and num_{edge}, where num_{job} is the number of retraining jobs generated at t and num_{edge} is the number of edges without retraining jobs (line 2). Then we greedily select the num edges with maximum computing capacity into $[J']$ in line 3. For each $j' \in [J']$, $j' = \text{argmax}_{j \in [J] \setminus [J'], p_j^r = 0}(f_j)$, where $\{p_j^r\}_{j \in J}$ is the proportion of resources allocated to retraining jobs in each edge. We allocate resources evenly from the remaining available resources for edges in $[J']$. The retraining resource of edge $j' \in [J']$ can be calculated as:

$$p^r_{j'} = \frac{\beta_k(\sum_j(f_j - p^r_j) - res_{min})}{(\alpha_t + \beta_k) * num * f_{j'}}, \forall j. \quad (2)$$

We denote $[R]$ as a set of possible weight ratios. Then the weight ratio between inference and retraining is selected from set $[R]$, aiming to find the ratio α_t/β_k that minimizes the overall latency at the current time slot (lines 4-9). After getting α_t and β_k, we can calculate the value of $p^r_{j'}$, $j' \in [J']$, and return p^r_j for each edge. Note that p^r_j is not fixed, it returns to 0 when there are no retraining jobs to be processed in edge j. Once all edges have been deployed with retraining jobs, subsequent jobs are distributed to an edge according to the A_{ret} algorithm, rather than allocating additional resources in the edge.

C. Inference Scheduling

Recall that p^r_j denotes the proportion of resources allocated to retraining jobs in edge j. We set f^t_j as the proportion of resources that can be utilized by inference tasks in edge j at t, $f^t_j = 1 - p^r_j$. The inference scheduling sub-problem can be formulated as:

$$\text{minimize} \quad P' = \sum_t \sum_i \sum_j y^t_{i,j} \tau^t_{i,j} \quad (3)$$

$$\text{s.t.} \quad \sum_i z^t_{i,j} \leqslant f^t_j, \ \ \forall j, \forall t, \quad (3a)$$

(1a),(1b),(1c),(1h)

Problem Transformation. For the inference tasks, we reformulate the scheduling problem into a Markov Decision Process (MDP). An MDP can be denoted by the tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\}$, where $\mathcal{S}$ indicates the state space, $\mathcal{A}$ denotes the action space, $\mathcal{P}$ represents the state transition probability, $\mathcal{R}$ denotes the reward, and $\gamma \in [0, 1]$ is the discount factor. In our scenario, the MDP can be defined as follows.

i) *State*. The controller observes the state information from edges and devices at each time slot t. The state is formulated as $\mathcal{S}_t = \{d^t_i, c^t_i, f^t_j\}_{i\in I, j\in J}$, where $\{d^t_i\}_{i\in I}$ and $\{c^t_i\}_{i\in I}$ respectively represent the data size and computing workload of the tasks generated by each device at t, and $\{f^t_j\}_{j\in J}$ indicates the proportion of computing resources of each edge that can be used for inference tasks.

ii) *Action*. Given the observed state $\mathcal{S}_t$, the controller determines the action $\mathcal{A}_t$ of the scheduling results for all inference tasks at time t, i.e., the edge selection $\{y^t_{i,j}\}_{i\in I, j\in J}$, the proportion of computing resources $\{z^t_{i,j}\}_{i\in I, j\in J}$ and the proportion of bandwidth $\{v^t_{i,j}\}_{i\in I, j\in J}$. Therefore, the action $\mathcal{A}_t$ can be defined as $\mathcal{A}_t = \{y^t_{i,j}, z^t_{i,j}, v^t_{i,j}\}_{i\in I, j\in J}$.

iii) *Reward*. Given the state and action at t, the controller will receive a reward $\mathcal{R}_t$ from the environment to evaluate the quality of action $\mathcal{A}_t$. Based on the objective of minimizing total latency, the reward function follows the principle that actions that mitigate latency are associated with higher rewards. Moreover, considering the constraint (1a), we develop a reward to impose penalties on actions that fail to meet the required SLO. The reward function is defined as follows:

$$\mathcal{R}_t = \frac{1}{I_t} \sum_{i\in I_t} \mathcal{R}_{i,t}, \text{where} \quad (4)$$

$$\mathcal{R}_{i,t} = \begin{cases} -\tau^t_i, & \text{constraint (1a) is satisfied,} \\ -(\tau^t_i + \psi), & \text{otherwise.} \end{cases}$$

$\mathcal{R}_{i,t}$ denotes the reward of task from device i at t, and ψ indicates the penalty factor. When the constraint (1a) is satisfied, the reward is the negative value of the task latency; otherwise, an augmented penalty ψ is appended to the reward. Note that the value of ψ must significantly exceed the average latency of tasks, enabling the controller to select actions that satisfy the constraint. The specific set of parameters is listed in Sec. V.

Design of RL Training Algorithm. Traditional dynamic programming methods are not effective in solving this MDP problem, while Reinforcement Learning (RL) has emerged as a promising approach for it. The selection of an RL network is crucial for fast training and exploration in MDP problems with high-dimensional states and multiple discrete actions. In our RL training algorithm, we exploit SAC-D to handle the inference scheduling MDP problem (3). The RL training algorithm includes an actor-critic architecture with an actor network, evaluative critic networks, and target critic networks. The actor network makes action decisions from the current state based on its policy π. In the output layer of the actor network, decision elements are addressed via discretization. Both evaluation critic networks and target critic networks use the clipped double Q-networks technique to mitigate the problem of Q-value overestimation and speed up training[?], and critic networks output the Q-value of each possible action rather than simply providing the action as input. The evaluation critic networks output a pair of Q-values (Q_{μ_1}, Q_{μ_2}) to evaluate the actor's actions, while the target critic networks calculate $(Q_{\mu'_1}, Q_{\mu'_2})$. For the algorithm, the goal of the controller is to find a policy π^* that maximizes the maximum entropy objective: $\pi^* = \arg\max_\pi \sum_{t\in T} \mathbb{E}_{(\mathcal{S}_t, \mathcal{A}_t)\sim\xi_\pi}[\gamma^t(\mathcal{R}_t + \lambda\mathcal{H}(\pi(.|\mathcal{S}_t)))]$, where λ denotes the temperature parameter that balances the reward and entropy, ξ_π indicates the distribution of trajectories induced by policy π, and $\mathcal{H}(\pi(.|\mathcal{S}_t))$ represents the entropy of the policy π at state $\mathcal{S}_t$. The transitions of the networks (i.e., $\{\mathcal{S}_t, \mathcal{A}_t, \mathcal{T}_t, \mathcal{S}_{t+1}\}_{t\in T}$) are stored into a replay buffer, which will be sampled randomly to train the networks later.

RL Network Parameter Update. During the training process, mini-batch transitions $\mathcal{U}$ are randomly sampled from the replay buffer to update the actor and critic networks. For a given transition $\{\mathcal{S}_e, \mathcal{A}_e, \mathcal{T}_e, \mathcal{S}_{e+1}\} \in \mathcal{U}$ the evaluative critic networks update the parameters independently by minimizing the loss function:

$$\mathcal{J}_Q(\mu_n) = \mathbb{E}_{(\mathcal{S}_e, \mathcal{A}_e)\sim\mathcal{U}}[\frac{1}{2}(\hat{Q}_e - Q_{\mu_n}(\mathcal{S}_e))^2], n = 1, 2, \quad (5)$$

with the target Q-values: $\hat{Q}_e = \mathcal{R}_e + \gamma(\min_{n=1,2} Q_{\mu'_n}(\mathcal{S}_{e+1}) - \lambda \log \pi_\delta(\mathcal{S}_{e+1}))$. Using the Stochastic Gradient Descent (SGD) method, the parameters $\{\mu_1, \mu_2\}$ can be optimized in the direction of $\nabla_{\mu_n}\mathcal{J}_Q(\mu_n)$.

To update the actor network, the objective function is:

$$\mathcal{J}_\pi(\delta) = \mathbb{E}_{\mathcal{S}_e\sim\mathcal{U}}[\pi_e(\mathcal{S}_e)^T[\lambda \log(\pi_\delta(\mathcal{S}_e)) - \min_{n=1,2} Q_{\mu_n}(\mathcal{S}_e)]], \quad (6)$$

and the temperature parameter λ can be updated by minimizing the entropy objective function:

$$\mathcal{J}(\lambda) = \pi_e(\mathcal{S}_e)^T[-\lambda \log(\pi_\delta(\mathcal{S}_e)) - \hat{H}], \tag{7}$$

where $\hat{H}$ is a constant and denotes the target entropy. Similar to the evaluative critic networks, the parameters of the actor networks δ and the temperature λ are updated by the SGD method.

Training Algorithm Details. The SAC-D based RL training algorithm for inference tasks is as follows:

Initialization. First, initialize the parameters of the actor network, evaluative critic networks and target critic networks in lines 1-2. Then initialize a replay buffer and several related hyperparameters in line 3. The algorithm requires E episodes of iterations. At the beginning of each episode, the environment is initialized, and the controller can receive the initial state in line 5.

Environment Step. At each time slot, the controller first obtains the information about the computing capacity of edges and update $\{f_j^t\}_{j\in J}$ in state $\mathcal{S}_t$ (line 7). Then the actor network utilizes the policy π_δ to make action decisions $\mathcal{A}_t = \{y_{i,j}^t, z_{i,j}^t, v_{i,j}^t\}_{i\in I, j\in J}$ based on the current state $\mathcal{S}_t = \{d_i^t, c_i^t, f_j^t\}_{i\in I, j\in J}$ and environmental information (line 8). In line 9, the actions are applied, and the controller receives the reward $\mathcal{R}_t$ and the next state $\mathcal{S}_{t+1}$ from the updated environment. In line 10, the experience tuple $\{\mathcal{S}_t, \mathcal{A}_t, \mathcal{R}_t, \mathcal{S}_{t+1}\}$ is stored into the replay buffer.

Network Training. In the training stage, a mini-batch of experience tuples U are randomly sampled from the replay buffer in line 11. Then, the parameters of the evaluative critic networks μ_n, the actor network δ and the temperature λ are updated separately by Eq. (5), Eq. (6) and Eq. (7) (line 12-14). Finally, the parameters of target critic networks μ_n' are updated using the soft-updating method, where ζ is the updating factor in line 15.

Inference Scheduling Algorithm. We employ an RL training algorithm to train an actor network $\pi_\delta(\mathcal{S})$ based on historical information of inference tasks. This actor network is utilized to make decisions regarding task offloading and resource allocation for current inference tasks. The inference scheduling algorithm (A_{inf}) is as follows: A_{inf} gets the state $\mathcal{S}_t$ from the information of tasks and edges (line 2). Then the policy π_δ is used to get the action $\mathcal{A}_t$ for tasks scheduling (line 3).

Algorithm 3 RL Training Algorithm

1: Initialize actor network $\pi_\delta(\mathcal{S})$ and evaluative critic networks Q_{μ_1}, Q_{μ_2} with parameters π, μ_1, μ_2;
2: Initialize target critic networks $Q_{\mu_1'}, Q_{\mu_2'}$ with $\mu_1' \leftarrow \mu_1$, $\mu_2' \leftarrow \mu_2$;
3: Initialize an empty experience replay buffer and hyperparameters γ, λ, ζ, ℓ_Q, ℓ_π, ℓ_λ, $\hat{\mathcal{H}}$;
4: **for** $episode = 1$ to E **do**
5: Initialize the environment, and receive the initial state $\mathcal{S}_1$;
6: **for** $t = 1$ to T **do**
7: Obtain the information about the inference resources of edges and update $\{f_j^t\}_{j\in J}$ in state $\mathcal{S}_t$;
8: Select action with the current policy $\mathcal{A}_t \sim \pi_\delta(\mathcal{S}_t)$;
9: Apply the actions $\mathcal{A}_t$ in the environment, receive the reward $\mathcal{R}_t$, and observe the next state $\mathcal{S}_{t+1}$;
10: Store $\{\mathcal{S}_t, \mathcal{A}_t, \mathcal{R}_t, \mathcal{S}_{t+1}\}$ into replay buffer;
11: Randomly sample a mini-batch of U experiences $\{\mathcal{S}_e, \mathcal{A}_e, \mathcal{T}_e, \mathcal{S}_{e+1}\} \in \mathcal{U}$ from replay buffer;
12: Update μ_1, μ_2 by minimizing the loss function in Eq. (5): $\mu_n \leftarrow \mu_n - \ell_Q \nabla_{\mu_n} J_Q(\mu_n), n = 1, 2$;
13: Update policy weight δ via gradient of Eq. (6): $\delta \leftarrow \delta - \ell_\pi \nabla_\pi J_\pi(\delta)$;
14: Update temperature λ via gradient of Eq. (7): $\lambda \leftarrow \lambda - \ell_\lambda \nabla_\lambda J(\lambda)$;
15: Update the parameters of target critic networks: $\mu_n' \leftarrow \zeta\mu_n' + (1-\zeta)\mu_n', n = 1, 2$.
16: **end for**
17: **end for**

Algorithm 4 Inference Scheduling Algorithm (A_{inf}) ,$\forall t$

Input: $B_j, f_j, p_j^r, [Tasks]_t, \forall j$
Output: $y_{i,j}^t, z_{i,j}^t, v_{i,j}^t, \forall i, \forall j$
1: Calculate $f_j^t = 1 - p_j^r$;
2: Update state $\mathcal{S}_t = \{d_i^t, c_i^t, f_j^t\}_{i\in[I], j\in[J]}$;
3: Get action from the actor network $\mathcal{A}_t \sim \pi_\delta(\mathcal{S}_t)$, $\mathcal{A}_t = \{y_{i,j}^t, z_{i,j}^t, v_{i,j}^t\}_{i\in[I], j\in[J]}$;

D. Retraining Scheduling

For retraining jobs, OINC will make decisions regarding job dispatching and resource allocation at each time slot.

Job Dispatching. To dispatch a job to the edge that brings the least increase to the total completion time, we simulate the algorithm in each edge. We assume edge j as the edge to which job k is dispatched. Set $s_{kj}(t) \triangleq l_k(t)/h_{kj}(t)$ as the weight density of job k at time slot t if it is dispatched to edge j. Here, $l_k(t)$ is the weight of job k, which is inversely proportional to the length of time until the deadline at time t. And $h_{kj}(t)$ is the remaining processing time of job k in edge j at time t, which can be calculated as:

$$h_{kj}(t) = \frac{w_k - \sum_{t'=a_t}^{t} \sum_j x_{k,j}^{t'} u_{k,j}^{t'}}{p_j^r f_j} \tag{8}$$

Assuming that no jobs will arrive in the future, the increase in total weighted completion time consists of three parts: 1) the weighted waiting time of job k, due to the other jobs with larger density than k (in the set of $Q_{kj}^1(t)$); 2) the weighted processing time of job k; 3) the weighted time of jobs with smaller density than k (in the set of $Q_{kj}^2(t)$). Then we can calculate the increase in completion time $C_{kj}(t)$ of job k in edge j:

$$\begin{aligned} C_{kj}(t) = \frac{1}{1+\epsilon}\{&l_k(t)\textstyle\sum_{k'\in Q_{kj}^1(t)} h_{k'j}(t) + l_k(t)h_{kj}(t) \\ &+\textstyle\sum_{k'\in Q_{kj}^2(t)} l_{k'}(t)h_{k'j}(t)\} \end{aligned} \tag{9}$$

$\frac{1}{1+\epsilon}$ is the speed augmentation factor. The dispatching algorithm is to assign the job k to edge j which minimizes $C_{kj}(t)$.

Job Processing Queue. Each edge maintains a queue of retraining jobs. In each time slot, each edge can process at most one retraining task. The edge determines the retraining jobs to be processed based on the scheduling algorithm, thereby ensuring that more urgent and weighted tasks are processed first. Let $Q_j(t)$ denote the jobs in edge j, which

have been dispatched but not completed at time t. Each edge sequences the jobs in $Q_j(t)$ according to their density $s_{kj}(t)$, and selects the job with the largest density to process at the current time. Note that, if a job k_1 has a higher density than another job k_2 at time t, k_1 will always have the higher density at a later time in the same edge. This avoids frequent switching of jobs processing in the edge.

Algorithm 5 Retraining Scheduling Algorithm (A_{ret}) ,$\forall t$

Input: $B_j, f_j, p_j^r, Q_j, [Jobs]_t, \forall j$
Output: $x_{k,j}^t, u_{k,j}^t, \forall j, \forall k$
1: **for** $Job_k \in [Jobs]_t$ **do**
2: /***Job Dispatching***/
3: Dispatch Job_k to edge j^*, $j^* = arg\min_{j\in[J]} C_{kj}(t)$;
4: $Q_{j*}(t) = Q_{j*}(t) \cup \{k\}$;
5: **end for**
6: **for** $j \in [J]$ **do**
7: /***Job Sequencing***/
8: Select Job_{k*} to process, $k^* = arg\min_{k\in Q_j(t)} s_{kj}(t)$;
9: $x_{k*,j}^t = 1, u_{k*,j}^t = p_j^r$;
10: Update $h_{k*j}(t)$ with Eq. (8);
11: **if** $h_{k*j}(t) \leqslant 0$ **then**
12: $b_{k*} = t$;
13: Remove Job_{k*} from edge j;
14: **end if**
15: **end for**

Retraining Scheduling Algorithm. For retraining jobs, A_{ret} first calls the dispatching algorithm, which distributes each job to an edge with the minimum increased completion time $C_{kj}(t)$ (lines 1-5). Then sequencing algorithm is invoked to select a retraining job to process for each edge at the current time (lines 8-9). Finally, the remaining process time $h_{kj}(t)$ is updated based on the decisions $x_{k,j}^t, u_{k,j}$ (line 10). The edge j removes Job_k if $h_{kj}(t) \leqslant 0$ (line 11-14).

E. Theoretical Analysis

First, we analyze the competitive ratio of the retraining scheduling algorithm (A_{ret}) by Theorem 1. The competitive ratio is defined as the largest possible ratio between the performance of the online algorithm and the offline optimal algorithm for any possible set of jobs. We use it as a metric to evaluate our online algorithm `OINC`. Then, we analyze the feasibility and time complexity of `OINC` in Theorem 2 and Theorem 3, respectively.

In order to analyze the competitive ratio, we offline the retraining job scheduling problem, which has all the information of jobs beforehand. We adopt the dual fitting technique to find the lower bound of the total weighted completion time of the offline problem.

i) *LP Relaxation*. We first derive an LP (Linear Programming) relaxation of the offline problem and prove that the LP relaxation has a lower bound. In the LP relaxation problem, time slots can be divided into different parts and allocated to process different jobs. We use m_{kj}^t to denote the proportion of time slot $[t, t+1]$ used by job k in edge j. For each job k, we define ϕ as:

$$\phi_k(\mathbf{m}) = \sum_{j,t} l_k m_{kj}^t \left(\frac{t - a_k}{h_{kj}} + \frac{1}{2}\right) \tag{10}$$

where $\mathbf{m}$ is the collection of all m_{kj}^t. The LP relaxation of the offline problem is shown as P_1:

$$\min \quad P_1 = \sum_k \phi_k(\mathbf{m}) \tag{11}$$

$$\text{s.t.} \quad \sum_j \sum_t \frac{m_{kj}^t}{h_{kj}} \quad \forall k, \tag{11a}$$

$$\sum_k m_{kj}^t \leqslant 1, \quad \forall j, \forall t, \tag{11b}$$

$$m_{kj}^t \geqslant 0, \quad \forall k, \forall j, t \geqslant a_k, \tag{11c}$$

The constraint (11a) means each job must be completed. Constraint (11b) guarantees that each edge can only process at most one retraining job at one time slot. We set F as the feasible solution of the offline problem, which can be uniquely translated to the feasible solution $\mathbf{m_F}$ of problem P_1.

The following lemma claims problem P_1 gives a lower bound of the offline problem.

Lemma 1. *The total weighted completion time of any feasible solution F for the offline problem is at least $\phi_k(\mathbf{m_F})$.*

Proof. For each job k, b_k is the completion time of k, and $m_{kj}^t = 0, \forall t > b_k$. When job k is processed from $b_k - h_{kj}$ to b_k, $\phi_k(\mathbf{m_F})$ is maximized.

$$\begin{aligned} \phi_k(\mathbf{m_F}) &\leqslant l_k \sum_{t'=1}^{h_{kj}} \left(\frac{b_k - a_k - t'}{h_{kj}} + \frac{1}{2}\right) \\ &= l_k(b_k - a_k - \frac{1}{2}) < l_k(b_k - a_k) \end{aligned} \tag{12}$$

□

Therefore, $\phi_k(\mathbf{m_F})$ is the lower bound of the total weighted completion time of feasible solution F in the offline problem.

ii) *Dual LP*. The dual LP of the LP relaxation problem is given as:

$$\max \quad P_2 = \sum_k \theta_k - \sum_{j,t} \eta_{jt} \tag{13}$$

$$\text{s.t.} \quad \frac{\theta_k}{h_{kj}} - \eta_{jt} \leqslant s_{kj}(t - a_k) + \frac{l_k}{2}, \quad \forall k, \forall j, t \geqslant a_k, \tag{13a}$$

$$\theta_k \geqslant 0, \eta_{jt} \geqslant 0, \quad \forall k, \forall j, \forall t, \tag{13b}$$

where θ_k and η_{jt} are the dual variables.

Theorem 1. ***(Competitive Analysis)*** *There exists a feasible solution to P2, such that the objective value (i.e. $\sum_k \theta_k - \sum_{j,t} \eta_{jt}$) is $\Omega(\epsilon F)$.*

Proof. We define θ_k^* as the increase of the total weighted completion time by job k, $\theta_k^* = \min_j C_{kj}(a_k)$. And η_{jt}^* is set as $\frac{1}{1+\epsilon}$ times of the total weights of the jobs in $O_j(t)$, $\eta_{jt}^* = \frac{1}{1+\epsilon} \sum_{k\in O_j(t)} l_k(t)$. $O_j(t)$ is a set of jobs, which have been dispatched to edge j but unfinished at time t.

It is obvious that $\sum_{j,t} \eta_{jt}^*$ is exactly equal to $\frac{F}{1+\epsilon}$, and $\sum_k \theta_k^*$ is equal to F. Therefore, we have $\sum_k \theta_k^* - \sum_{j,t} \eta_{jt}^* = \frac{\epsilon}{1+\epsilon} F$.

Lemma 2. *By setting $\theta_k = \theta_k^*/2$ and $\eta_{jt} = \eta_{jt}^*/2$, we get a feasible solution to P2.*

Proof. We fix a edge j and a job k, then we set t as the arriving time of k and $t' \geqslant t$. We need to prove:

$$\frac{\theta_k^*}{h_{kj}} - \eta_{jt'}^* \leqslant s_{kj}(t'-t) \tag{14}$$

All terms in (14) will not be affected by the jobs arriving after t, except for $\eta_{jt'}^*$ which will increasing because of more jobs in $O_j(t')$. So we can assume there is no new job released after job k.

We suppose edge j' will process job k' at time t'. t_a denotes the corresponding time of k' appearing in $Q_{kj}^1(t) \cup Q_{kj}^2(t)$ (arriving time), then we can have:

$$\frac{(1+\epsilon)\theta_k^*}{h_{kj}} \leqslant s_{kj}(t'-t)(1+\epsilon) + \eta_{jt'}^*(1+\epsilon) \tag{15}$$

Divide both sides of (15) by $2(1+\epsilon)$, we have:

$$\frac{\theta_k^*}{2h_{kj}} - \frac{\eta_{jt'}^*}{2} \leqslant \frac{s_{kj}(t'-t)}{2} \leqslant s_{kj}(t'-t) \tag{16}$$

Therefore, it shows that $\frac{\theta_k^*}{2}$ and $\frac{\eta_{jt'}^*}{2}$ can satisfy the inequality (14). □

Thus, the objective function $\Sigma_k \theta_k - \Sigma_{j,t}\eta_{jt} = \frac{1}{2}(\Sigma_k \theta_k^* - \Sigma_{j,t}\eta_{jt}^*) = \frac{\epsilon}{2(1+\epsilon)}F = \Omega(\epsilon F)$ □

Given that the objective value of the dual problem is a lower bound of the total weighted completion time of the offline problem. It can be proved that the online scheduling algorithm is $O(\frac{1}{\epsilon})$-competitive with $(1+\epsilon)$-speed augmentation.

Theorem 2. ***(Feasibility Analysis)*** *Our online algorithm framework* OINC *produces a feasible solution to the optimization problem (1).*

Proof. We first prove that the RL training algorithm (Algorithm 3) can produce feasible solutions for sub-problem (3). In the output layer of the actor network, three softmax layers are utilized to individually normalize the decision variables $\{y_{i,j}^t, z_{i,j}^t, v_{i,j}^t\}_{i\in[I],j\in[J]}$. For device i, the value of them is set to 0, when i has not generated inference tasks at t. By setting these, constraints (3a), (1b) and (1c) can be satisfied. And the penalty set in the reward can ensure the constraint (1a) will not be violated. For Algorithm 1, the constraint (1d) can be met because the dispatching algorithm distributes a retraining job to only one edge in line 14. In lines 22-24, Algorithm 1 checks the remaining process time of each job, and removes the job which is completed, so the constraint (1e) is met for every job. Therefore, it can be proved that algorithm OINC returns a feasible solution. □

Theorem 3. ***(Time Complexity)*** *OINC runs in polynomial time, with time complexity* $O(KJR + T(I + J + IJ))$.

Proof. We first analyze the time complexity of Algorithm 2. The edge selection in line 3 checks the available computing capacity for each edge, which takes $O(J)$ steps. The for loop in lines 4-8 calculates the total latency based on different weight ratios, which tasks $O(RJ)$ steps at most. Therefore, the time complexity of Algorithm 2 is $O(RJ)$.

As for Algorithm 1, the update in line 5 tasks $O(K + JT)$ steps considering the loop of time slots. The for loop in lines 12-16 dispatches jobs to edges, which takes $O(KJ)$. Similarly, the for loop in lines 17-26 selects a job to process in each edge, which also takes $O(KJ)$. And the inference scheduling decisions are made by the actor network in lines 29-30. With different parameters, we have fixed the number of layers and the number of neurons in the intermediate layers of the actor network, while the number of neurons in the input layer, $(2I + J)$, and the number of neurons in the output layer, $(3IJ)$, both depend on the number of devices and edges. Therefore, we approximate the time for online inference of the actor network as $O(T(I + J + IJ))$.

In summary, the time complexity of OINC is $O(KJR + T(I + J + IJ))$. □

V. Experiment Evaluation

A. Evaluation Settings

System Setup. The number of edges is set within $[10, 15]$ (default $J = 12$), and the number of devices varies between 20 to 50 with an increment of 10 (default $I = 40$). Similar to [32], the computing capability of edges is $[5, 9]$ GHz, and the wireless bandwidth is set to $[2, 3]$ MHz. Each edge has a different computing capacity and bandwidth. The channel gain $g_{i,j}^t$ is modeled as $140.7 + 36.7 * \log_{10}(dis) + 4$, where dis denotes the transmission distance between edges and devices [33], which is randomly set from 0.15 to 0.45 in km [8]. The transmission power ρ_i is set within $[80, 100]$ mW [34], and the Gaussian noise is set to -40 dbm/Hz.

Inference Tasks. Similar to [35], we adopt the 24-hour traces of the Alibaba production cluster to simulate the generation and workload of tasks from different devices [36]. Specifically, we randomly select I different jobs from the traces within 10 seconds, and expand each of them in reasonable proportions to the total runtime of the system, according to the real-world situation. The data size of inference tasks is within $[0.8, 1.6]$ MB, while the computing workload is set from 100 to 200 Megacycles. The SLO of tasks varies according to the devices. For retraining jobs, the workload is determined by the batch size, the number of retraining epochs, and the size of the retraining dataset. We set the batch size as 8 [19] and the default size of the retraining set as 1024. We consider 1 second as a single time slot, and the number of total time slots T is set as 200.

DNNs and Datasets. Similar to [19], we demonstrate OINC's effectiveness using five compressed DNNs: ResNet18 [37], MobileNetV2 [38], ShuffleNet [39], TinyYOLOv2 [40], AlexNet [41]. As explained in Sec.III-A, We use ResNeXt-101 [42] as the golden model to label drift data in the cloud. Both inference and retraining stages utilize the widely used ImageNet dataset [43] and a specialized dataset CORe50 [44], which is designed for continuous learning to simulate data drift.

RL Networks. In the RL training algorithm, the actor network and critic networks are all four-layer neural networks, which consist of an input layer, an output layer, and two hidden layers. The number of neurons in the hidden layers is 1024 and 512, respectively. The penalty factor of reward ψ is set to

TABLE I: Parameter Setting of the RL training algorithm

Parameter	Value	Parameter	Value
Number of episodes E	5000	Number of steps T	200
Replay buffer size	10000	Mini-batch size U	100
Learning rate $\ell_Q, \ell_\pi, \ell_\lambda$	0.0001	Discount factor γ	0.99
Temperature initial λ	1.0	Target entropy $\hat{H}$	-log$(1+\iota)$
Soft update factor ζ	0.01	Optimizer	Adam
Hidden layer act.	ReLU	Actor output act.	Softmax

3. Other parameters of RL networks are listed in TABLE I , where ι denotes the output dimension of the actor network.

Baselines. To evaluate the performance of the OINC algorithm, we compare it with the following six baselines.

- *Ekya* [19]: *Ekya* is a heuristic algorithm that makes inference and retraining scheduling jointly. In this approach, inference tasks and retraining jobs are treated as the same kind of tasks when scheduling.
- *Kalmia* [45]: *Kalmia* is a specialized algorithm designed for scheduling urgent and non-urgent tasks to guarantee deadlines and improve throughput. We assume inference tasks as urgent tasks and retraining jobs as non-urgent tasks.
- *Greedy*: Greedy algorithm sorts inference tasks according to SLOs and greedily selects an edge for tasks in order. Then it uses a convex optimization solver to determine resource allocation.
- *Liu* [8]: *Liu* algorithm is a state-of-art task scheduling algorithm for edge inference tasks. It offloads tasks based on a primal-dual approximation algorithm and allocates resources by utilizing the Lagrangian multiplier method.
- *Dedas* [13]: *Dedas* algorithm is a scheduling algorithm focused on the deadline of jobs, which dispatches and schedules jobs greedily to satisfy deadlines as much as possible.
- *Zhang* [14]: *Zhang* algorithm schedules jobs in batches and configures a weight for each job in the batch.

Among the six baselines, Ekya and Kalmia are two algorithms to jointly schedule both inference tasks and retraining jobs. They are used to compare with OINC for performance evaluation. Among the remaining four baselines, Greedy and Liu are inference task scheduling algorithms, while Dedas and Zhang are retraining job scheduling algorithms. We conduct ablation experiments using these four algorithms to demonstrate the significance of our two sub-algorithms (A_{inf} and A_{ret}) in Sec. V-B.

B. Evaluation Results

Evaluation Metrics. Considering the limitations of edge resources and the requirements of inference task SLO and retraining job deadlines, not all tasks and jobs can be successfully completed. Therefore, we consider the following metrics to evaluate the performance of OINC. i) Weighted average latency of all completed inference tasks and retraining jobs. ii) Success rate, which is calculated as the number of successfully completed tasks over the total number of tasks, including inference and retraining. Inference tasks that meet the SLOs in terms of latency and retraining jobs that are completed within the deadlines can both be considered as successfully completed tasks.

Fig. 2: The convergence performance of RL.

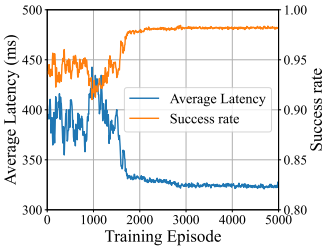

Fig. 3: The convergence performance details of RL.

Convergence of RL. The convergence performance of our RL training algorithm is presented in Fig. 2. The light blue region represents the standard deviation of the reward. It can be observed that the reward value rises gradually as the number of training episodes increases until it reaches a relatively stable value. The results show that the RL training algorithm converges after parameter iterations for 2000 episodes. Specifically, we plot the success rate and the average latency under each epoch in Fig. 3. It illustrates that the average latency decreases and the success rate rises gradually as the number of episodes grows, which further proves that the RL training algorithm has a good convergence effect, i.e., small average latency with a high success rate.

TABLE II: Average latency for inference and retraining with different algorithms

Algorithms / Latency (s)	Inference	Retraining	Sum
OINC (A_{inf}&A_{ret})	**0.250**	**96.209**	**96.459**
Ekya	0.277	101.691	101.968
Kalmia	0.262	98.372	98.634
Greedy & A_{ret}	0.479	100.684	101.163
Liu & A_{ret}	0.271	100.684	100.955
A_{inf} & Dedas	0.256	101.286	101.542
A_{inf} & Zhang	0.256	144.326	144.582

Performance Detail. In order to verify the performance of OINC, we compare it with two baselines: Eyka and Kalmia. The average inference latency, average retraining completion time, and the sum of them are listed in TABLE II. Eyka algorithm allocates resources to both inference tasks and retraining jobs without distinguishing between them. Additionally, once the resources are allocated, they remain unchanged. Kalmia algorithm prioritizes the allocation of resources to inference tasks and allocates remaining resources in each time slot to retraining jobs. However, the OINC algorithm takes into account both the SLOs of inference tasks and the deadlines of retraining jobs simultaneously. The results show that OINC can effectively reduce the average latency of the entire system compared with the Eyka algorithm and Kalmia algorithm.

Ablation Result. We further combine the inference scheduling algorithm (A_{inf}) and the retraining scheduling algorithm (A_{ret}) with four comparison algorithms respectively, which aim to separately explore the impact of the two scheduling algorithms on latency. The ablation experiment result is also listed in TABLE II. When A_{inf} and A_{ret} are used with

other algorithms, we simply divide the computing resources into two parts based on the weight of inference tasks and retraining jobs. Then we use scheduling algorithms to allocate resources to inference tasks and retraining jobs respectively. The results illustrate that the lack of any one of the two scheduling algorithms, A_{inf} and A_{ret}, will increase the latency of inference and retraining. In addition, the experiment also demonstrates that our reservation algorithm (Alg. 2) performs better than just allocating resources according to weights.

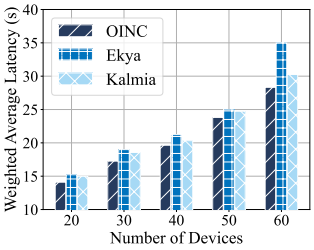

Fig. 4: The weighted average latency with different number of devices.

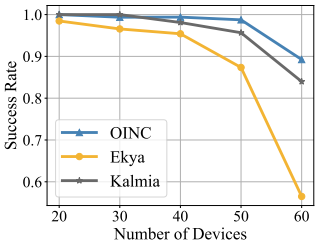

Fig. 5: The success rate with different number of devices.

Impact of Device Number. Fig. 4 and Fig. 5 respectively illustrate the weighted average latency and the success rate of three algorithms in terms of the number of devices, when the number of edges is fixed. The results show that the weighted average latency increases gradually as the number of devices increases, while the success rate decreases. This is attributed to the increase in the number of inference tasks as the number of devices increases. With a constant number of edges, indicating a fixed total computing resources, the computing latency increases and the success rate decreases. As evident from the figures, even with limited edge resource capacity, OINC enhances the success rate by up to 33.2% while reducing the weighted average latency by up to 23.7%.

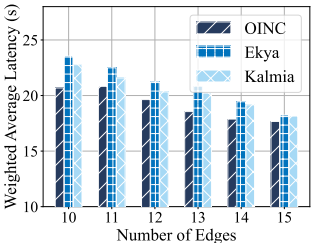

Fig. 6: The weighted average latency with different number of devices.

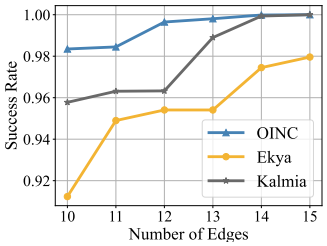

Fig. 7: The success rate with different number of edges.

Impact of Edge Number. Fig. 6 and Fig. 7 respectively investigate the weighted average latency and the success rate of three algorithms in terms of the number of edges. As the number of edges increases, the total capacity of computing resources in the system also increases, resulting in lower latency and higher success rates for both inference and retraining tasks. As can be seen from Fig. 4, to obtain the same weighted average latency as the OINC algorithm under 10 edges, the Ekya algorithm requires 14 edges and Kalmia requires 12 edges. Similarly, Ekya and Kalmia need additional edges, along with additional computing resources, to achieve an equivalent success rate as OINC.

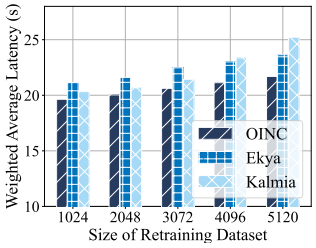

Fig. 8: The weighted average latency with different retraining workload.

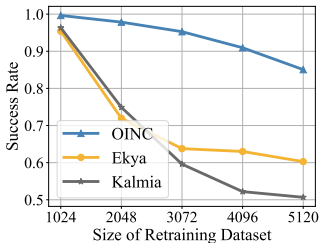

Fig. 9: The success rate with different retraining workload.

Impact of Retraining Workload. Fig. 8 and Fig. 9 respectively visualize the weighted average latency and success rate as the size of the retraining dataset varies, which is directly proportional to the workload of retraining jobs. As the size of the retraining dataset increases, both the Ekya and Kalmia algorithms tend to disregard the impact of increased retraining workload on scheduling priorities for retraining and inference, resulting in an increase in latency and a significant decrease in success rate. In contrast, OINC consistently produces low average latency and maintains a high success rate through the explicit consideration of the weight of inference and retraining based on their workload in the reservation algorithm (Alg. 2). Specifically, OINC can boost success rate by up to 35.6% while reducing latency by 21%.

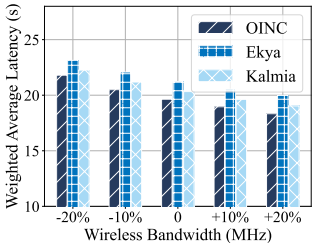

Fig. 10: The weighted average latency with different wireless bandwidth.

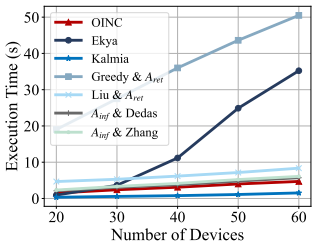

Fig. 11: The execution time with different algorithms.

Impact of Wireless Bandwidth. Fig. 10 shows the weighted average latency as the wireless bandwidth varies. For instance, a value of -20% in the figure denotes that the wireless bandwidth of each edge decreases to 0.8 times its initial value. The figure illustrates the stability of our algorithm when faced with changing wireless transmission environments. OINC performs better than other algorithms with different wireless bandwidths.

Execution Time. Fig. 11 compares the execution time of the algorithms as the number of devices increases. We run all the algorithms 5 times and calculate the average as the execution time. The results show that our algorithm OINC runs at a slightly slower pace than the Kalmia algorithm, but faster than other algorithms. The total duration of the system is set as 200 seconds. The total execution time of OINC is less than 5 seconds, which is 2.5% of each time slot. Considering the low latency and high success rate of OINC in limited computing resources and heavy workload, our execution time is deemed acceptable compared with the other six algorithms.

VI. Conclusion

Online edge DNN inference with continuous learning brings new challenges to scheduling both inference tasks and retaining jobs on edges. To this end, we propose a new online algorithm, OINC, to simultaneously schedule inference tasks and retraining jobs in edge networks, aiming to minimize the weighted total latency of these two types of tasks. OINC first reserves resources for inference and retraining according to the workload. Next, OINC leverages an RL algorithm to offload inference tasks and designs a preemptive algorithm for scheduling retraining jobs. Both theoretical analysis and experiment results validate the efficiency and the superiority of OINC.

References

[1] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[2] A. Hekmati, P. Teymoori, T. D. Todd, D. Zhao, and G. Karakostas, "Optimal mobile computation offloading with hard deadline constraints," *IEEE Transactions on Mobile Computing*, vol. 19, no. 9, pp. 2160–2173, 2020.

[3] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Comp. and Comm.*, vol. 23, no. 4, p. 11–20, 2020.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *arXiv: Computer Vision and Pattern Recognition*, 2015.

[5] D. Maltoni and V. Lomonaco, "Continuous learning in single-incremental-task scenarios," *Neural Netw.*, vol. 116, no. C, p. 56–73, 2019.

[6] X. Yin, X. Yu, K. Sohn, X. Liu, and M. Chandraker, "Feature transfer learning for face recognition with under-represented data," in *Proc. of CVPR*, 2019.

[7] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning," in *Proc. of SoCC*, 2018.

[8] H. Liu, X. Long, Z. Li, S. Long, R. Ran, and H.-M. Wang, "Joint optimization of request assignment and computing resource allocation in multi-access edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1254–1267, 2023.

[9] B. Chen, A. Bakhshi, G. E. A. P. A. Batista, B. Ng, and T.-J. Chin, "Update compression for deep neural networks on the edge," *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 3075–3085, 2022.

[10] L. Zhang, G. Gao, and H. Zhang, "Towards data-efficient continuous learning for edge video analytics via smart caching," in *Proc. of SenSys*, 2023.

[11] R. T. Mullapudi, S. Chen, K. Zhang, D. Ramanan, and K. Fatahalian, "Online model distillation for efficient video inference," in *Proc. of ICCV*, 2019.

[12] E. Aleksandrova, C. Anagnostopoulos, and K. Kolomvatsos, "Machine learning model updates in edge computing: An optimal stopping theory approach," in *Proc. of ISPDC*, 2019.

[13] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *Proc. of IEEE INFOCOM*, 2019.

[14] Q. Zhang, R. Zhou, C. Wu, L. Jiao, and Z. Li, "Online scheduling of heterogeneous distributed machine learning jobs," in *Proc. of Mobihoc*, 2020.

[15] Y. Li, T. Zeng, X. Zhang, J. Duan, and C. Wu, "Tapfinger: Task placement and fine-grained resource allocation for edge machine learning," in *Proc. of IEEE INFOCOM*, 2023.

[16] N. Wang, R. Zhou, L. Jiao, R. Zhang, B. Li, and Z. Li, "Preemptive scheduling for distributed machine learning jobs in edge-cloud networks," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 8, pp. 2411–2425, 2022.

[17] W. Chu, P. Yu, Z. Yu, J. C. Lui, and Y. Lin, "Online optimal service selection, resource allocation and task offloading for multi-access edge computing: A utility-based approach," *IEEE Transactions on Mobile Computing*, vol. 22, no. 7, pp. 4150–4167, 2023.

[18] Z. Liang, Y. Liu, T.-M. Lok, and K. Huang, "Multi-cell mobile edge computing: Joint service migration and resource allocation," *IEEE Transactions on Wireless Communications*, vol. 20, no. 9, pp. 5898–5912, 2021.

[19] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *Proc. of NSDI*, 2022.

[20] W. Fan, L. Zhao, X. Liu, Y. Su, S. Li, F. Wu, and Y. Liu, "Collaborative service placement, task scheduling, and resource allocation for task offloading with edge-cloud cooperation," *IEEE Transactions on Mobile Computing*, pp. 1–18, 2022.

[21] T. Liu, S. Ni, X. Li, Y. Zhu, L. Kong, and Y. Yang, "Deep reinforcement learning based approach for online service placement and computation resource allocation in edge computing," *IEEE Transactions on Mobile Computing*, vol. 22, no. 7, pp. 3870–3881, 2023.

[22] Y. Liu, Y. Mao, Z. Liu, F. Ye, and Y. Yang, "Joint task offloading and resource allocation in heterogeneous edge environments," in *Proc. of IEEE INFOCOM*, 2023.

[23] N. Eshraghi and B. Liang, "Joint offloading decision and resource allocation with uncertain task computing requirement," in *Proc. of IEEE INFOCOM*, 2019.

[24] H. Jiang, X. Dai, Z. Xiao, and A. Iyengar, "Joint task offloading and resource allocation for energy-constrained mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 22, no. 7, pp. 4000–4015, 2023.

[25] E. Moro and I. Filippini, "Joint management of compute and radio resources in mobile edge computing: A market equilibrium approach," *IEEE Transactions on Mobile Computing*, vol. 22, no. 2, pp. 983–995, 2023.

[26] A. Bhattacharjee, A. D. Chhokra, H. Sun, S. Shekhar, A. Gokhale, G. Karsai, and A. Dubey, "Deep-edge: An efficient framework for deep learning model update on heterogeneous edge," in *Proc. of ICFEC*, 2020.

[27] J. a. Gama, I. Żliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, 2014.

[28] V. M. A. Souza, A. R. S. Parmezan, F. A. Chowdhury, and A. A. Mueen, "Efficient unsupervised drift detector for fast and high-dimensional data streams," *Knowledge and Information Systems*, vol. 63, pp. 1497 – 1527, 2021.

[29] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "icarl: Incremental classifier and representation learning," in *Proc. of CVPR*, 2017.

[30] Y. Chen, Z. Liu, Y. Zhang, Y. Wu, X. Chen, and L. Zhao, "Deep reinforcement learning-based dynamic resource management for mobile edge computing in industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 7, pp. 4925–4934, 2021.

[31] F. M. C. Forum, "5g vehicular communication technology," 2017.

[32] Y. Ma, W. Liang, M. Huang, W. Xu, and S. Guo, "Virtual network function service provisioning in mec via trading off the usages between computing and communication resources," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2949–2963, 2022.

[33] Y. Dai, D. Xu, S. Maharjan, and Y. Zhang, "Joint computation offloading and user association in multi-task mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 12, pp. 12 313–12 325, 2018.

[34] L. Li, T. Q. Quek, J. Ren, H. H. Yang, Z. Chen, and Y. Zhang, "An incentive-aware job offloading control framework for multi-access edge computing," *IEEE Transactions on Mobile Computing*, vol. 20, no. 1, pp. 63–75, 2021.

[35] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *Proc. of USENIX ATC*, 2019.

[36] Alibaba production cluster trace data. [Online]. Available: Available:https://github.com/alibaba/clusterdata

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of CVPR*, 2016.

[38] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. of CVPR*, 2018.

[39] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proc. of CVPR*, 2018.

[40] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *Proc. of CVPR*, 2017.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Proc. of NIPS*, 2012.

[42] H. Wang, A. Kembhavi, A. Farhadi, A. L. Yuille, and M. Rastegari, "Elastic: Improving cnns with dynamic scaling policies," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2253–2262, 2018.

[43] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. of CVPR*, 2009.

[44] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," in *Proc. of CoRL*, 2017.

[45] Z. Fu, J. Ren, D. Zhang, Y. Zhou, and Y. Zhang, "Kalmia: A heterogeneous qos-aware scheduling framework for dnn tasks on edge servers," in *Proc. of IEEE INFOCOM*, 2022.