CoEdge: A Cooperative Edge System for Distributed Real-Time Deep Learning Tasks

Zhehao Jiang[†]
The Chinese University of Hong Kong
Hong Kong SAR, China
jz018@ie.cuhk.edu.hk

Neiwen Ling[†]
The Chinese University of Hong Kong
Hong Kong SAR, China
lingnw@link.cuhk.edu.hk

Xuan Huang
The Chinese University of Hong Kong
Hong Kong SAR, China
1155136647@link.cuhk.edu.hk

Shuyao Shi
The Chinese University of Hong Kong
Hong Kong SAR, China
ss119@ie.cuhk.edu.hk

Chenhao Wu
The Chinese University of Hong Kong
Hong Kong SAR, China
chenhaowu@link.cuhk.edu.hk

Xiaoguang Zhao
The Chinese University of Hong Kong
Hong Kong SAR, China
xgzhao@cuhk.edu.hk

Zhenyu Yan
The Chinese University of Hong Kong
Hong Kong SAR, China
zyyan@cuhk.edu.hk

Guoliang Xing[*]
The Chinese University of Hong Kong
Hong Kong SAR, China
glxing@ie.cuhk.edu.hk

ABSTRACT

Recent years have witnessed the emergence of a new class of *cooperative edge systems* in which a large number of edge nodes can collaborate through local peer-to-peer connectivity. In this paper, we propose *CoEdge*, a novel cooperative edge system that can support concurrent data/compute-intensive deep learning (DL) models for distributed real-time applications such as city-scale traffic monitoring and autonomous driving. First, CoEdge includes a hierarchical DL task scheduling framework that dispatches DL tasks to edge nodes based on their computational profiles, communication overhead, and real-time requirements. Second, CoEdge can dramatically increase the execution efficiency of DL models by batching sensor data and aggregating the inferences of the same model. Finally, we propose a new edge containerization approach that enables an edge node to execute concurrent DL tasks by partitioning the CPU and GPU workloads into different containers. We extensively evaluate CoEdge on a self-deployed smart lamppost testbed on a university campus. Our results show that CoEdge can achieve up to 82.32% reduction on deadline missing rate compared to baselines.

CCS CONCEPTS

• **Computer systems organization** → **Sensor networks**; • **Computing methodologies** → **Self-organization**; • **Software and its engineering** → *Software design engineering*.

[†] Co-primary Authors, [*]Corresponding Author.

IPSN '23, May 9–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0118-4/23/05…$15.00
https://doi.org/10.1145/3583120.3586955

KEYWORDS

Smart City, Edge Computing, Distributed Deep Learning System, Real-time Scheduling, Edge Containerization

ACM Reference Format:
Zhehao Jiang[†], Neiwen Ling[†], Xuan Huang, Shuyao Shi, Chenhao Wu, Xiaoguang Zhao, Zhenyu Yan, and Guoliang Xing[*]. 2023. CoEdge: A Cooperative Edge System for Distributed Real-Time Deep Learning Tasks. In *The 22nd International Conference on Information Processing in Sensor Networks (IPSN '23), May 9–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3583120.3586955

1 INTRODUCTION

Edge computing has emerged as a major distributed computing paradigm in which services are pushed from the cloud to the edge of the network to enable a wide range of Internet of Things (IoT) applications [37]. Moreover, edge devices are increasingly running Deep Learning models to support advanced data-intensive applications such as autonomous driving [22] and smart traffic infrastructures [4]. Due to the limited computing resource, edge devices typically execute a compressed set of DNN models [10, 20, 50] or offload partial DNN workloads via the connectivity to the cloud [15, 17, 54, 55].

However, a key bottleneck of current edge systems lies in the reliance on the powerful centralized Internet infrastructure. Recently, with the advances of communication technology [24, 49], there emerges a new class of *cooperative edge systems*[1] where a large number of edge nodes are connected with each other through local peer-to-peer connectivity [48, 51]. Compared to the conventional edge architecture, cooperative edge systems yield superior scalability, as they allow edge devices to be deployed over a large geographic region without relying on the computing or communication capability of the Internet. For instance, a smart roadside infrastructure system [25] can achieve real-time intelligent traffic monitoring and support autonomous driving [13, 36] through the collaboration of sensor-enabled lampposts connected via local

[1] *Cooperative Edge* is also referred to as *Fog Computing System* [8] in some contexts.

wireless networks. Moreover, such a system can serve as a shared infrastructure for a multitude of different geo-distributed applications developed and deployed by different service providers.

Despite the aforementioned advantages, a cooperative edge system must address several major challenges. First, because of the inherently distributed nature, the edge nodes must handle heavy and often uneven workloads. For instance, a smart lamppost typically needs to execute multiple DNN models to process the data streams of multi-modal sensors such as LiDAR, radar, and thermal. In a smart factory [6, 38], a high-throughput assembling pipeline can pose a heavier workload on the edge nodes in proximity than other nodes. Therefore, as a fundamental requirement, a cooperative edge system must not only support concurrent DNN tasks on each edge node, but also must collaborate efficiently to support advanced data-intensive and real-time applications. Moreover, mobile and geo-distributed applications such as autonomous driving are likely built in different system environments. To address such heterogeneity in execution environments, mainstream virtualization technologies such as containers are increasingly available on the network edge [29]. However, the current containerization technology lacks the key support for concurrent DNN execution. For instance, multiple containers cannot access GPU resources on the same edge simultaneously.

To address these challenges, we propose a new cooperative edge system named CoEdge, which supports distributed real-time applications through efficient collaboration among different edge nodes and concurrent execution of DL tasks on a single edge node. The design of CoEdge is based on a *hierarchical DL task scheduling framework*, which implements *global task dispatching* and *batched DNN execution* on local edge nodes. The global task dispatcher allocates DL tasks to edge nodes and achieves efficient edge collaboration by jointly considering the network bandwidth among edge nodes and real-time performance for DL tasks on each edge node. Moreover, we design a local batched DNN execution mechanism for efficient resource utilization on each edge node. Different from general batch processing for model training, we aim to use batch to process real-time inference tasks. CoEdge can batch the inference tasks that use the same model as much as possible, resulting in better GPU resource utilization and real-time performance. Lastly, we propose a novel approach called *GPU-aware concurrent DL containerization*, which provides an isolated execution environment for each DL task while ensuring concurrent model inferences can share the same GPU. Meanwhile, this GPU-aware containerization structure can well support the local batch execution.

We implement CoEdge based on mainstream industrial platforms KubeEdge[47], ROS2 [23], and TensorRT [31]. We conduct extensive experiments on an indoor testbed and a real-world outdoor smart lamppost testbed of 6 nodes. Our results show that CoEdge can support distributed DL tasks to achieve satisfactory real-time performance. Moreover, compared to several state-of-the-art baselines, CoEdge can reduce the deadline missing rate up to 80% without sacrificing any accuracy.

The contributions of this paper are summarized as follows:

(1) We propose a hierarchical DL task scheduling framework that supports efficient distributed DL tasks on the cooperative edge system.
(2) We design a batched DNN execution mechanism that utilizes GPU resources efficiently and optimizes real-time performance for concurrent DNN execution on the edge nodes.
(3) We devise a GPU-aware concurrent DL containerization approach to support concurrent DL tasks with diverse execution environments on the edge node.
(4) We implement CoEdge and evaluate the performance through extensive experiments on a real-world smart lamppost testbed and an indoor testbed. Our results validate key advantages of CoEdge in supporting distributed real-time DL applications.

2 RELATED WORK

2.1 DL Task Offloading

To meet the stringent timing requirements for DL tasks on edge devices, an efficient approach for accelerating the DNN model is to offload the compute workload to the cloud or other edge devices. Neurosurgeon [15] predicts the latency and power consumption of each layer and automatically partitions DNN at the layer granularity for workload offloading. EdgeML [55] dynamically adjusts the partition points based on the runtime communication bandwidth with a reinforcement learning algorithm. ENGINE [7] adopts a greedy method to determine which tasks should be executed locally or sent to the cloud to minimize the energy cost. However, these methods only consider a single client and server, while distributed DL tasks typically require a coordinated execution among multiple nodes.

To balance the workloads among multiple edge devices, the work in [30] optimizes the overall response time for user requests under an edge-cloud architecture by formulating an integer linear programming problem. Their approach offloads all data on edge devices to the cloud, which leads to high latency for edge-cloud communication and cannot fully utilize the computing resources of edge devices. Dedas [27, 28] is an online deadline-aware task dispatching and scheduling mechanism. Dedas offloads tasks to the cloud when the edge devices cannot satisfy tasks' real-time requirements. This work assumes that resource-rich servers are deployed close to the source of the data and the end devices, which is not held in cooperative edge platforms. What's more, cloud offloading may lead to privacy disclosure.

2.2 Concurrent DL Tasks on the Edge

Several works are focused on optimizing the real-time performance of concurrent DL tasks on the edge. DART [46] employs a pipeline-based scheduling architecture with data parallelism for real-time DNN inference requests. RT-mDL [20] optimizes the DL task execution on edge platform to meet their diverse real-time/accuracy requirements by joint model scaling and scheduling. BlastNet [19] proposes a novel model inference abstraction and designs a dynamic cross-processor scheduler to support efficient DNN model inference on heterogeneous CPU-GPU platforms. However, the above approaches focus on the real-time performance of concurrent DL tasks on a single edge device, and do not consider the model inferences from other edge devices in a cooperative manner.

Moreover, current works do not address the compatibility issues when deploying multiple DNN models on a set of shared edge platforms. For example, the pre-/post-processing of multiple tasks

may require different versions of the software packages like NumPy, SciPy, and Numba. To support an isolated environment for DL tasks, a common approach is encapsulating DL tasks in separate containers [42, 45]. However, such a containerization technique cannot be directly adapted for concurrent DL tasks due to the limitations of edge GPU platforms, such as the lack of virtualization support. A recent work [34] partitions containers into two parts, one shared container for executing codes run in the same environments and one individual container. Although such an approach can support the containerization of multiple edge applications, it is not designed for concurrent DL tasks on GPU-accelerated platforms.

3 APPLICATIONS

There exists a range of distributed real-time applications that adopt DL models for data/compute-intensive tasks, such as smart traffic infrastructure [4], autonomous driving [22], smart ports and factories [6, 38]. Here, we describe *smart roadside infrastructure* as an example to illustrate the possible application scenarios of CoEdge.

Fig. 1 shows a smart roadside infrastructure that consists of multiple smart sensor-enabled lampposts. Each lamppost equips with multiple modalities of sensors such as thermal cameras, mmWave radars, and LiDARs to provide smart traffic monitoring, infrastructure-assisted autonomous driving, and other smart services. Nevertheless, edge nodes on lampposts usually have limited computing and power budgets. For instance, without an overhaul upgrade, existing city roadside lighting infrastructure only provides a fixed power supply of tens of Watts. These limitations are not unusual for large-scale edge systems such as smart lampposts that must rely on existing power and network facilities. In the following, we discuss the common characteristics and requirements of such applications.

3.1 Real-time and concurrent DNN execution

Many applications supported by smart roadside infrastructure have stringent timing requirements. For instance, to achieve real-time vehicle tracking, the edge nodes on smart lampposts must detect passing vehicles and communicate with nearby edge nodes within seconds. In infrastructure-assisted autonomous driving [36], the edge nodes can even be required to process sensor data and transmit the results to vehicles in real time. Moreover, as a shared infrastructure, each edge node is typically required to execute a multitude of these applications concurrently. For instance, a node may run multiple DNN models for vehicle detection from its adjacent nodes at the same time. The limited on-device resources and tight real-time requirements hence pose a major challenge for the design of cooperative edge systems.

3.2 Geo-distributed, uneven workloads

The services and workloads among multiple lampposts are inherently unevenly distributed due to the diverse characteristics of sensors and the complexity of road conditions. For example, thermal cameras, mmWave radars, and LiDARs typically have a sensing range of 10 to 500 meters. As a result, these sensors are often deployed according to road/traffic conditions and budgets, which results in highly diverse and dynamic data/compute workloads on different edge nodes.

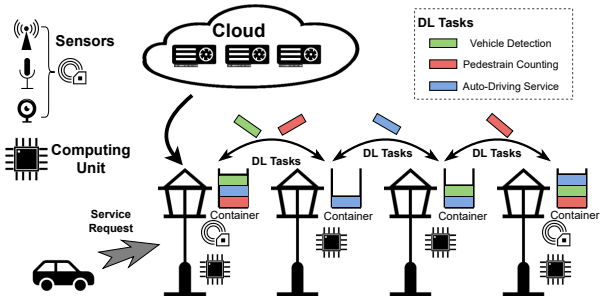


Figure 1: Task scenario of distributed real-time support for roadside infrastructure.

3.3 Diverse development/runtime environments

As a shared infrastructure, the edge node must support a diverse set of mobile and geo-distributed applications, which are likely developed by different service providers in different software/hardware environments. For instance, different authorities may develop and deploy services for security surveillance, vehicle monitoring, tracking, and autonomous driving applications. To address such heterogeneity in system environments, mainstream virtualization technologies such as containers are widely adopted by existing edge systems [29]. However, the current containerization technology lacks the key support for concurrent DNN execution on edge GPU. For instance, multiple containers cannot access the same edge GPU simultaneously, which cannot support concurrent DNN execution with efficient resource utilization.

4 SYSTEM DESIGN

4.1 Overview of CoEdge

To support distributed real-time DL tasks among multiple edge devices, we propose a new cooperative edge system named CoEdge. CoEdge integrates global task dispatching and local batched real-time DL execution into a hierarchical DL task scheduling framework. Moreover, we also devise a novel GPU-aware concurrent DL containerization approach that provides an isolated execution environment for each task while ensuring efficient utilization of computing resources on the edge platform. A bird-eye view of CoEdge is shown in Fig. 2. CoEdge has three major components, i.e., global task dispatcher, batched real-time DNN execution, and GPU-aware concurrent DL containerization, which work together as a cooperative edge system for executing distributed real-time DL tasks.

In the hierarchical DL task scheduling framework, we design a global task dispatcher to balance the DL workloads among different edge devices and a local batched DNN execution mechanism to improve the real-time performance of DL tasks on each edge node. The global task dispatcher jointly considers the network bandwidth among edge nodes and real-time performance for DL tasks on each edge node. A key challenge here is to estimate the real-time performance (i.e., response time from task release to completion) of each DL task under different strategies. However, the execution time of distributed DL tasks is challenging to estimate due to the dynamics of the network condition and runtime resources on the

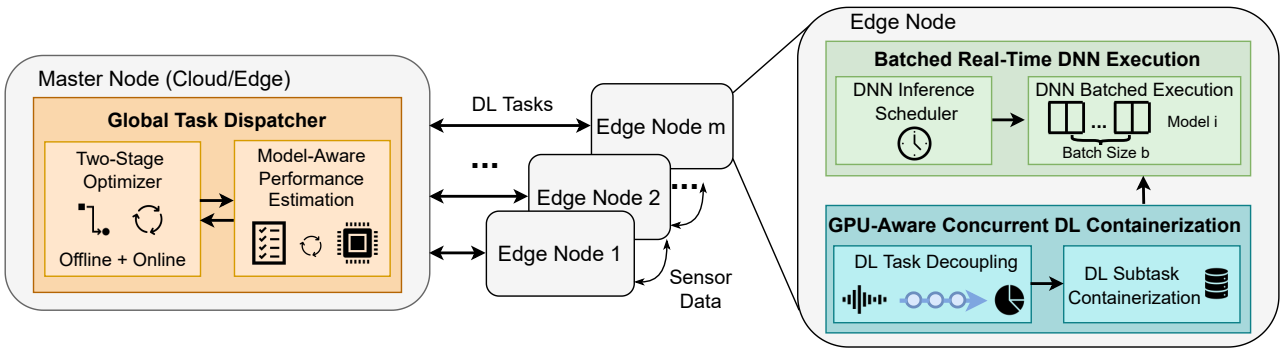


Figure 2: The system architecture of CoEdge: the global task dispatcher optimizes the allocation of distributed DL tasks, batched real-time DNN execution batches the DNN model inference of different DL tasks, and the GPU-aware concurrent DL containerization supports different execution environments for distributed DL tasks.

edge platform. To address this challenge, CoEdge adopts a two-stage optimization strategy during the task dispatch. First, CoEdge finds an initial allocation strategy based on the offline profiled execution time. Then, CoEdge optimizes the allocation strategy online based on the real-time performance measured at runtime. Meanwhile, our global task dispatcher also takes the task similarity into account to encourage batched execution on each node.

On each edge node, we batch the model inferences that use the same model as much as possible while ensuring their real-time requirements. This design is motivated by the key observation that batch processing of the same model will increase the GPU spatial utilization and thus decrease the average inference time for each DNN model. When multiple DL tasks adopt the same DNN model, the data from different sources could be concatenated for batched inference, which avoids executing the same model repeatedly and hence reduces resource usage. A challenge in exploiting batch processing here is that a larger batch size corresponds to higher GPU utilization, which may lead to deadline missing due to the prolonged overall execution time. CoEdge addresses this issue by profiling the DNN model inference time with different batch size settings offline and then optimizes the batch size with an online DNN inference scheduler. Our DNN inference scheduler maximizes the benefit of batched model inference while meeting the real-time requirements for each task by controlling the execution order and batch size of each DNN model inference. Under such a mechanism, we can utilize GPU resources more efficiently and achieve better real-time performance.

As a mainstream virtualization technology, the container is essential for supporting distributed DL tasks, which are typically developed in different system environments. However, a key challenge is that current containerization technology lacks support for the edge GPU [33]. Specifically, multiple containers cannot access the GPU on the same edge node simultaneously, which leads to inefficient resource utilization due to the sequential execution of DL tasks. We design a GPU-aware concurrent DL containerization approach to provide an isolated execution environment for each DL task while ensuring concurrent model inferences can share the same GPU. Our containerization mechanism also supports batched DNN execution of concurrent tasks. With our GPU-aware DL containerization structure, DL tasks with different environmental dependencies can be executed simultaneously on the same GPU. Specifically, CoEdge decomposes each DL task into CPU-based pre/post-processing and GPU-based DNN inference subtasks. Then, CoEdge encapsulates each CPU subtask into a separate container and all the DNN inference subtasks into one container. This way, we can fully utilize computing resources and resolve the incompatible computing environments of different DL tasks.

4.2 Hierarchical DL Task Scheduling

As described in Section 3, to support concurrent execution of real-time geo-distributed DL tasks, the system needs to coordinate their execution among different edge nodes and optimize their execution on a single edge node. Considering the unbalanced workload caused by the sensor distribution, the geographical distribution, and the lack of computing resources of a single edge node, we propose a hierarchical DL task scheduling framework, consisting of a task dispatcher on the master node and local schedulers on edge nodes. The framework accounts for the location of sensors and network conditions to minimize the communication delay caused by data transmission. Moreover, it effectively uses the computation resources and communication bandwidth among edge devices.

4.2.1 Overview of hierarchical DL task scheduling. Unlike the traditional distributed task scheduling which only balances the workloads among nodes [44], we design a hierarchical DL scheduling mechanism for the distributed DL task execution, which fully considers the workload balance among nodes and the actual execution efficiency of each node. As shown in Fig. 3, our hierarchical DL scheduling framework consists of two parts: a task dispatcher on the master node and batched DNN executions on multiple edge nodes. The task dispatcher allocates the DL tasks to edge nodes based on the estimation of their real-time performance. Each edge node runs the allocated DL tasks with the batched real-time DNN execution mechanism. It also monitors the actual real-time performance of each DL task and communicates with the master node for run-time allocation strategy optimization.

4.2.2 Task dispatcher on the master node. To coordinate distributed DL task execution among different edge devices, we design a task

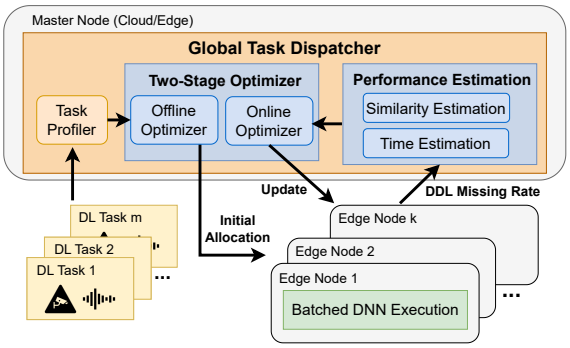


Figure 3: Hierarchical DL task scheduling framework of CoEdge

dispatcher on the master node. It contains a two-stage optimizer, which allocates tasks based on the offline profiled execution time and then optimizes the allocation based on the online real-time performance. We also design a similarity-based strategy to maximize the benefit of local batched DNN execution where more DL tasks with the same DNN model are executed on the same node.

Suppose there are totally k_{max} edge nodes $N = \{n_1, n_2, ..., n_{k_{max}}\}$. And there are m_{max} DL tasks denoted as $D = \{\tau_1, \tau_2, ..., \tau_{m_{max}}\}$. Each DL task can be represented as $\tau_i = \{n_i^{src}, n_i^{exe}, s_i, M_i, d_i, P_i\}$, which includes data source node n_i^{src}, its execution node n_i^{exe}, source data size s_i, DNN model type M_i, task deadline d_i and task priority P_i. We denote the set of tasks deployed on the node n_k as $\mathcal{S}_k = \{\tau_m | X(\tau_m, n_k) = 1, \forall m\}$. $X(\tau_m, n_k)$ is an indicator function, and $X(\tau_m, n_k) = 1$ indicates the task τ_m is deployed on the node n_k. We denote the end-to-end latency of DL task τ_m executed on edge node n_k as $T_{m,k}$, where $T_{m,k} = T_{m,k}^{com} + T_{m,k}^{exe}$, i.e., the sum of the communication time and the execution time. We quantify the real-time performance of each DL task using the deadline missing rate. We assume that each DL task releases jobs periodically. The deadline missing rate is the percentage of jobs that missed their deadlines among all jobs of a periodic DNN inference task, which can be denoted by $MR(\cdot) = (N^{overdue\_job} + N^{dropped\_job})/N^{all\_jobs}$. $N^{overdue\_job}$ refers to the number of jobs whose end-to-end latency $T_{m,k}$ exceeds its deadline d_m.

Our goal is to minimize the tasks that are already allocated on the edge nodes as well as those that have not been deployed, because they also need to be deployed if there exist available resources. We formulate the total missing rate of all deployed tasks as $\sum_{m=1}^{m_{max}} \sum_{k=1}^{k_{max}} MR_{m,k} \times X(\tau_m, n_k)$, and the total missing rate of not deployed tasks as $\sum_{m=1}^{m_{max}} \sum_{k=1}^{k_{max}} (1 - X(\tau_m, n_k))$. The goal is to minimize the total missing rate of the deployed tasks and not deployed tasks as defined below:

$$\begin{aligned} \max_{\forall m,k} \quad & \sum_{m=1}^{m_{max}} \sum_{k=1}^{k_{max}} (1 - MR_{m,k}) \times X(\tau_m, n_k) \\ \text{s.t.} \quad & \sum_{k=1}^{k_{max}} X(\tau_m, n_k) \leq 1, \quad \forall m, \\ & MR_{m,k} \leq \epsilon_i, \quad \forall X(\tau_m, n_k) = 1, \end{aligned} \tag{1}$$

where ϵ_i is the tolerable deadline missing rate of τ_i.

We now analyze the communication time of the DL task. In real-world settings, communication bandwidth is influenced by a variety of factors, including network topology, signal strength, and interference. Real-time bandwidth prediction is still a challenging problem. There exist many methods for bandwidth prediction based on complex models [26] [35]. However, these approaches incur excessive computation on edge devices. In this work, we measure the bandwidth periodically and calculate the communication delay based on the measured bandwidth. The communication time for DL task τ_m is defined in Eq. 2.

$$T_{m,k}^{com} = \frac{s_m}{B(n_{src}^m, n_k)} \tag{2}$$

where $B(n_{src}^m, n_k)$ denotes the bandwidth between edge node n_{src} and n_k.

Another part of end-to-end latency is attributed to the execution of DNN models. There exist several solutions for predicting DNN inference time based on generalized prediction models. Some studies [21] use the model's FLOPS or/and MAC as proxies for latency. However, these metrics can not precisely capture the actual delay, because they do not account for the difference of platforms and network structure [9]. Other solutions like nn-Meter [52] detect the fusion rules of a target platform and generate individual predictors of the kernel. These methods can predict the inference time of a single model on a specific platform but are not applicable for concurrent DL task execution. To accurately model concurrent DNN inference time, many factors must take into account, such as blocking time, scheduling policy, the number of concurrent tasks, etc. To address this challenge, we first estimate the execution time of a task based on its execution condition in the whole task set and then measure the real-time performance online. The estimated execution time for DL task τ_m can be represented as

$$T_{m,k}^{exe} = \sum_{j=1}^{|\mathcal{S}_k|} \frac{p_m \times T_{avg}(j)}{p_j} + C \tag{3}$$

where p_j denotes the period of task τ_j, $T_{avg}(j)$ denotes the average model inference time of task τ_j, C is a sum of pre/post-processing time.

However, due to the dynamics in resource usage, the estimation of communication delay and execution delay obtained with the above method may be inaccurate at runtime. Hence, we adjust the end-to-end latency when the actual missing rate is larger than the estimated missing rate, which can be represented as $T_{m,k}^{exe} = (1 + \alpha) \times T_{m,k}^{exe}$. We will illustrate how to adjust this parameter in the following.

To achieve the objective defined in Eq. 1, we design a task dispatcher as shown in Fig. 3. The task dispatcher is based on an iterative profiling approach. It tries to deploy as many tasks as possible on edge nodes while minimizing the deadline missing rate. Given a set of edge nodes and DL tasks, CoEdge allocates the DL tasks from high to low priority. In the first round of allocation, CoEdge estimates the communication time (Eq. 2) and the execution time (Eq. 3) based on the offline profiled data. For a new DL task τ_m to be allocated to the node n_k with the task set $\mathcal{S}_k$, we first check whether its estimated time $T_{m,k}$ on this node can meet the expected

deadline. If the deadline can be met, we calculate its similarity with the current DL tasks on this node based on the cosine similarity as shown in Eq. 4,

$$Sim(\tau_m, \mathcal{S}_k) = cosine\ similarity(EV(\tau_m), EV(\mathcal{S}_k)) \quad (4)$$

where $EV(\cdot)$ denotes the number tuple of the model type set. For example, if there is a total of 3 model types, DL task τ_m has the model of type 2, $EV(\tau_m) = (0, 1, 0)$. If there are 2 tasks with model type 1, 3 tasks with model type 3 in $\mathcal{S}_k$, $EV(\mathcal{S}_k) = (2, 0, 3)$. We choose the node with the highest task similarity and the lowest latency as the node to host the current DL task. CoEdge repeats the above allocation process until there is no new DL task that can meet its deadline.

Then, each node records the run-time deadline missing rate of each task when running tasks according to the allocation strategy. If the measured deadline missing rate cannot meet the real-time requirements of the DL task, the dispatcher will remove the task with the same and lower priority and increase the parameter α. This is because the online deadline missing rate may be caused by the deterioration of network conditions or a shortage of resources on edge devices. We update the measured bandwidth to cope with such performance degradation caused by bandwidth deterioration. Since resource shortage on the edge platform will influence the actual execution speed of the DL task, we decrease the α to bring the estimated execution time closer to the actual value.

4.3 Batched DNN Execution

In this section, we will introduce how to efficiently utilize the GPU resources to meet real-time requirements for each DL task. Previous work [20] proposes a GPU task packing mechanism to increase the GPU spatial utilization for real-time DNN execution. However, this mechanism packs the DL tasks with different execution times, which may lead to synchronization issues and unnecessary waiting time. As we discussed in Section 3, for distributed DL tasks, we need to process the data from different edge nodes with the same DNN model. Hence, we propose to batch the data and process them simultaneously, which increases the GPU spatial utilization and avoids the synchronization overhead among different DL tasks. As shown in Fig. 4, batch processing can effectively decrease the average inference time of each model. For example, the YOLO model with an input size of 160×160 can decrease its average DNN model inference time by a factor of 2 with a batch size of 3. The batch size represents the number of samples (e.g., image frames) that will be passed through the network at one time. Motivated by this insight, we design a new batched DNN execution mechanism to run the application in a batch manner while ensuring that each task can complete in time. Specifically, we design a DNN inference scheduler to schedule the execution order of different models to meet the real-time requirements for each DL task. Meanwhile, we also adjust the batch size dynamically for each model inference to achieve more efficient GPU resource utilization. A larger batch size corresponds to higher GPU utilization but may lead to deadline missing for some tasks. To address this problem, we profile the DNN model inference time with different batch size settings offline and adjust the batch size online.

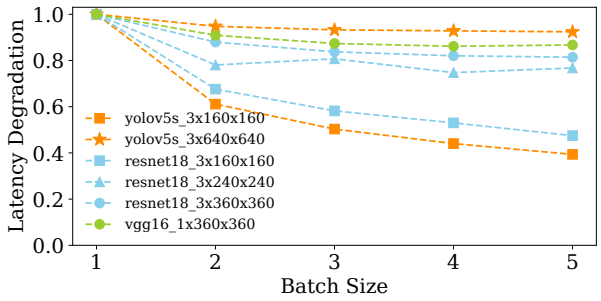


Figure 4: Batch Efficiency (evaluated under TensorRT on the edge platform NVIDIA Jetson TX2).

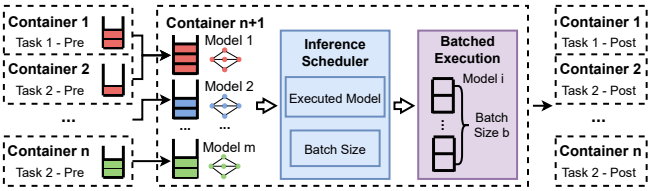


Figure 5: Batched DNN execution on the single edge node.

As we will discuss in Section 4.2, CoEdge pushes the preprocessed data from the application queue to the model inference queue. The DL tasks that adopt the same DNN model are put in the same DNN model inference queue. For each model inference, we launch an independent process and preload the model into memory before execution. Thus, this approach avoids the huge overhead caused by model loading and inference initialization.

To achieve efficient GPU resource utilization and meet the real-time requirements for each DL task, we design a new DNN inference scheduling algorithm to control the execution order and batch size of each DNN model inference. Our scheduler first determines which model to be executed according to the urgent level of each model inference. Specifically, as shown in Eq. 5, we first calculate the minimum deadline for all the jobs in each model inference queue. $DDL_{i,g(i)}$ denotes the absolute deadline of $g(i)^{th}$ job in the model inference queue for model i. $g(i)$ is the total number of all the jobs in the current model inference queue i. Then, we choose the model that has the minimum deadline in the model inference queue for execution.

$$Model\ Index = \underset{\mathbf{i}}{\operatorname{argmin}}\{min_{j=1,2...,g(i)}\{DDL_{i,j}\}\} \quad (5)$$

Our scheduler then determines the batch size for each model inference. The main goal here is maximizing the benefit brought by batched model inference while meeting the real-time requirements for each task. We achieve this by profiling the execution time for each setting of batch size offline and determining the batch size online. For offline profiling, we profile the setting of batch size from 1 to 10. This is sufficient to accommodate the dynamics at runtime because the actual deadline for each job is unlikely to exceed 10 times the execution time of single model inference. At runtime, we choose the maximum batch size that can meet the real-time requirements of all the jobs. Specifically, as shown in Eq. 6, we compare

the estimated completion time (i.e., current time plus estimated execution time) and the deadline of each job to estimate whether the selected batch size can meet the real-time requirement of each job.

$$Batch\ Size = \underset{b}{\text{argmax}}\{CurTime + ExeTime(i,b) < min_{i,2,\ldots,I,j=1,2\ldots,g(i)}\{DDL_{i,j}\}\} \quad (6)$$

where $ExeTime(i,b)$ denotes the profiled execution time of model i with batch size b. Our scheduler will update the execution strategy (i.e., model index and batch size) when the current model finishes execution.

Once the model index i and batch size b for the current model execution is determined, the scheduler will fetch data of batch size b from the model inference queue. The DNN executor then batches the fetched data and feeds it to the model i for inference. Under this batched DNN execution mechanism, CoEdge can effectively utilize the GPU resource and thus better meet timing requirements for concurrent real-time DNN inferences.

4.4 GPU-aware Concurrent DL Containerization

Containers have been widely used on edge systems for supporting different DL tasks with isolated execution environments. As we discussed in Section 3, real-world DL tasks often have different requirements on execution environments, which need to be executed independently in separate containers. However, current low-power edge devices (e.g., NVIDIA Jetson TX2, AGX Xavier) usually have only one GPU and do not support GPU virtualization. In this case, if we encapsulated each DL task in a separate container, it would be impossible to run multiple DL tasks at the same time, since different containers cannot access the GPU on the same edge node simultaneously. As a result, multiple DL tasks must be executed sequentially. However, considering DL tasks have both workloads on CPU and GPU, sequential execution of DL tasks will lead to inefficient resource utilization.

A naive way to address this problem is to encapsulate all DL tasks in one container and bind the container to the GPU on the edge device. However, this approach leads to two major issues in system deployment. First, the execution environments of different DL tasks may conflict with each other. As a result, some DL tasks may fail to execute. For example, a model developed in 2022 may be built under PyTorch 1.13, while a model developed in 2018 is likely to be built under PyTorch 0.4. Second, a single container that must execute multiple DL tasks may have complex environmental dependencies, which makes it infeasible to the deployment of additional DL tasks. For example, existing applications in a single container all rely on NumPy1.14 for array processing, while the new DL tasks have a package of Numba0.55.2 which needs NumPy<1.23, >=1.18. As a result, it will be infeasible for the deployment of the new DL task.

This section will introduce our design of GPU-aware concurrent DL containerization, which provides an isolated execution environment for each DL task, and ensures that all model inferences can share the same GPU resource. Our containerization mechanism is carefully designed to support the local batch execution as discussed in Section 4.3. With our mechanism, DL tasks with different environmental dependencies can be executed simultaneously on the same GPU. This enables different DL tasks with the same DNN model can be executed in a batched manner.

An end-to-end DL task contains not only the model inference on the GPU but also the execution of pre-processing and post-processing on the CPU, which accounts for a considerable portion of end-to-end task delay [20]. For most DL tasks, only the model inference will occupy the GPU, while pre-/post-processing mostly use CPU. Therefore, we split each DL task into two to three parts and encapsulate them into separate Docker containers, where all DNN model inferences of different DL tasks share a container that is bound to the GPU on the edge device. This mechanism enables more efficient utilization of heterogeneous computing resources on edge devices and resolves the incompatible software dependencies of different DL tasks at the same time.

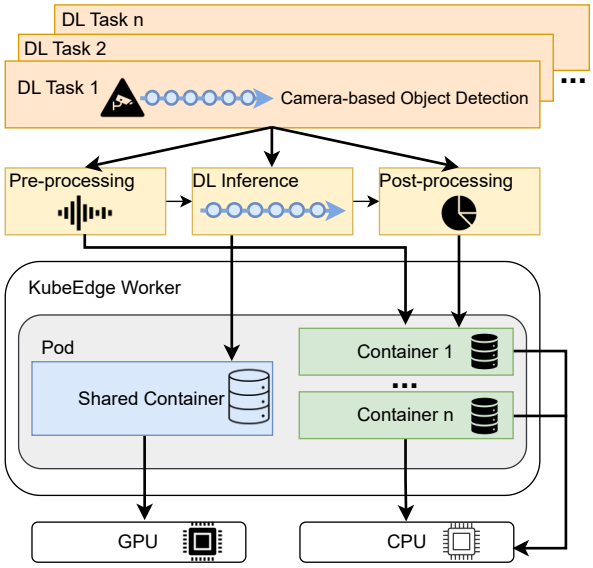


Figure 6: GPU-aware concurrent DL containerization on the single edge node.

As shown in Fig. 6, each DL task will be divided into three parts: pre-processing, DNN inference, and post-processing. The pre-processing and post-processing of one task are assigned to two containers, while all DNN inferences of different tasks share the same container, which can directly access the GPU resource of the edge device. Meanwhile, to eliminate the differences in dependent environments of different DNN models, all DNN models are converted to open neural network exchange format (ONNX) [3], and are managed by the local scheduler, which will be discussed in Section 4.3. As a result, all DL tasks can be isolated from each other while being processed concurrently in real time.

Although GPU-aware containerization can make the DL tasks isolated, it brings challenges in inter-container communication. Different containers use the same host OS kernel and communicate using one of the three mechanisms: named pipes, UNIX domain socket, and shared memory. Named pipes are a FIFO method, typically used for very small and high-frequency message communications between two processes. However, large chunks of sensor data

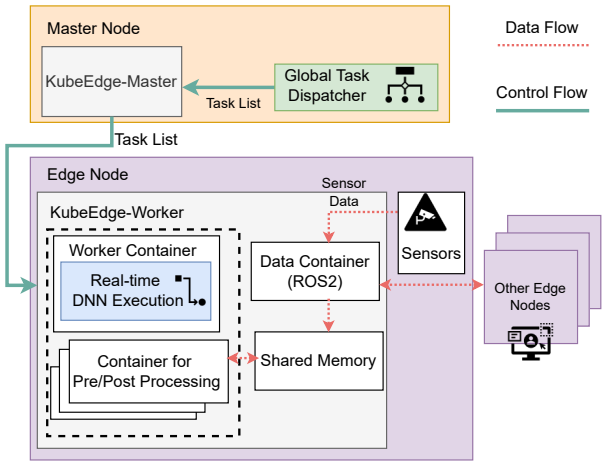


Figure 7: The software implementation of CoEdge on KubeEdge.

often need to be passed between containers when they are executed concurrently. The isolated containers may communicate with each other via the UNIX domain socket. However, the socket does not support multicast or broadcast, which is required in our design to achieve communication between a single node and multiple nodes.

Hence, we adopt the shared memory as the buffer of a queue to achieve inter-container communication. Specifically, each queue is connected by peer-to-peer communication, and only transmits the address of the data on the shared memory, which can reduce the copying times of the data.

5 IMPLEMENTATION

We have designed and implemented CoEdge based on KubeEdge [47], ROS2 [23] and TensorRT [31]. We choose KubeEdge since it is a lightweight system to facilitate the management of edge containerized applications, and is gaining the support of the developer community. To manage heterogeneous data sources, CoEdge reads the sensor data through ROS2, which is widely used for sensor management. We also convert the DNN models into the TensorRT format to speed up the DNN inference on edge platforms. Fig. 7 shows the architecture of CoEdge implementation.

Master node. Initially, on the master node, CoEdge collaborates with the KubeEdge master for task dispatching. Once the global task dispatcher updates the task allocation strategy, it will dispatch the corresponding containers that encapsulate the sub-tasks (i.e., pre/post-processing and model inference) to the edge node. Meanwhile, the corresponding task parameters such as the IDs of data source nodes will also be dispatched to the target edge node.

Edge node. On the edge node, CoEdge runs concurrent DNN tasks on the received containers. Those containers are managed by KubeEdge. CoEdge also launches a container for data management based on ROS2. In this container, CoEdge reads the sensor data with the sensor driver and creates sensor publishers for dispatching data. CoEdge also launches sensor subscribers to establish connections with other sensor publishers for data fetching. For example, if the task on Node2 needs the sensor data from Node3, the sensor subscriber on Node 2 will receive the data dispatched from the sensor publisher on Node3. CoEdge stores the fetched data into shared memory for task processing as discussed in Section 4.4.

Table 1: DNN tasks for evaluation.

DNN Task Type	Sensor Type	DNN Model	Dataset	Model Input Size
Object Detection-1	Camera	YOLOv5s	CoCo [18]	(3x640x640)
Object Detection-2	Camera	YOLOv5s	CoCo	(3x160x160)
Image Classification-1	Thermal	VGG19	Teledyne FLIR ADAS[11]	(3x160x160)
Image Classification-2	Camera	ResNet18	CIFAR10[16]	(3x360x360)

Table 2: Computing platforms of edge and master nodes.

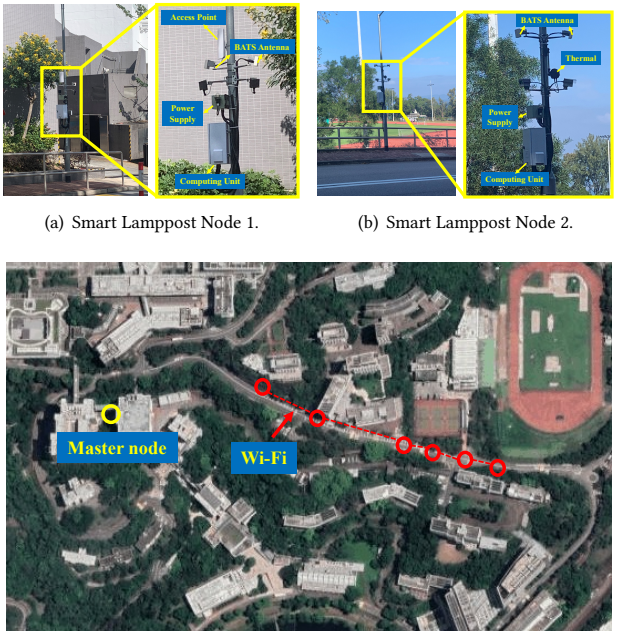
Platform	GPU	CPU	Memory	Storage
NVIDIA Jetson TX2 (Edge Node)	256-core Pascal	2-core ARM Denver + 4-core ARM A57	8GB	500 GB
Indoor Master Node	NVIDIA GeForce 960M	4-core Intel	8 GB	500 GB
Outdoor Master Node	N.A.	Intel Xeon GOLD 5117 (14 Core, 2.0GHz, 19.25M)	5x64 GB	2x8 TB

6 EVALUATION

In this section, we first discuss the experimental setup in Section 6.1, and then describe evaluation metrics and baselines in Section 6.2. Second, we validate CoEdge on a real-world smart lamppost testbed and an indoor testbed in Section 6.3 and 6.4, respectively. In addition, we evaluate the performance of the batched DNN execution (Section 4.3) and the GPU-aware concurrent DL containerization (Section 4.4), in Section 6.5 and Section 6.6, respectively.

6.1 Experiment Setup

6.1.1 Smart Lamppost Testbed. We deployed CoEdge on an outdoor smart lamppost network we installed on a university campus for performance evaluation. The testbed consists of 12 lamppost nodes and a master node. Table 2 shows the specifications of the edge and master nodes. Each smart lamppost is installed with an NVIDIA Jetson TX2 computing board [32] with a MIC-720-AI [1] waterproof enclosure with auxiliary sockets and an extra 450 GB solid-state drive (SSD). The communication between edge nodes is realized using a multi-hop wireless network that achieves an average throughput of around 80 Mbps by integrating a network coding algorithm with 802.11ac. The bandwidth between two edge nodes is measured by iperf3, a popular speed test tool for TCP, UDP, and SCTP. Each edge node is also equipped with a 4G cellular



(a) Smart Lamppost Node 1. (b) Smart Lamppost Node 2.

(c) The installation locations of nodes.

Figure 8: Our outdoor smart lamppost testbed on campus.

dongle to communicate with the master node located in a building. Fig. 8 shows examples of two lamppost nodes and the deployment layout.

6.1.2 Lab Testbed. We set up a lab testbed to evaluate CoEdge under controlled network conditions. We use three NVIDIA Jetson TX2 computing boards [32] as the edge nodes, which can provide 1.33 TFLOPS of computing power with a maximum power consumption of only 20 watts. A laptop PC with an Intel 4-core CPU and an NVIDIA GeForce 960M GPU serves as the master node. Table 2 shows the specifications. We connect all edge nodes and the master node to a PoE switch, which can configure the network bandwidth during experiments.

6.1.3 DL Tasks. We evaluate CoEdge with four DNN tasks using two different types of sensors and three DNN models. Table 1 shows the details of DNN tasks. In particular, we apply two different models for image classification on the camera, i.e., YOLOv5s [14] and ResNet 18 [12]. Note that we set two image sizes of YOLOv5s for the two IP cameras of the indoor testbed, which evaluate CoEdge with different sensor input sizes and workloads. We use VGG19 [39] for object recognition from the thermal camera images. The number of frames per second of each task is set to 1/deadline. In addition, we transform each model into the open neural network exchange format (ONNX) [3] and compile them using TensorRT [31] to accelerate the execution.

6.2 Evaluation Metrics

We use two metrics to evaluate CoEdge's real-time performance in distributed DL tasks.

Deadline Missing Rate. We quantify the real-time performance of each DL task with/without our system using the *deadline missing rate*, which is a widely-used metric for real-time performance evaluation [5, 20]. Specifically, the deadline missing rate is defined in Section 4.2, which denotes the percentage of jobs that missed their deadlines among all jobs of a periodic DNN inference task. We measure the deadline missing rate within a period of 120 s.

End-to-end Latency. We use *end-to-end latency* to quantify the execution efficiency of each DL task. It is defined as the total delay between the launch and the completion of a task, including the communication time between edge nodes, pre/post-processing time, DNN model inference time, and the blocking time caused by resource contention.

6.3 Results on Smart Lamppost Testbed

We implement CoEdge and evaluate its end-to-end system performance on the smart lamppost testbed deployed on the campus. As shown in Fig. 8(a)-8(b), each smart lamppost is equipped with a small embedded platform (Nvidia Jeston TX2) and sensors such as thermal cameras. The main purpose of this system is to assist the campus security office with traffic management and provide an open testbed for various research projects. Specifically, we used three major functions in our experiments: real-time traffic monitoring, pedestrian recognition, and vehicle recognition, which are implemented using YOLO, VGG, and ResNet, respectively [2, 53]. The real-time traffic monitoring task detects the vehicle on the road for traffic monitoring, and the vehicle recognition task distinguishes whether there is a vehicle in the field of view. We set the priorities of three tasks in the following order (from high to low): YOLO-based real-time traffic monitoring, VGG-based pedestrian recognition, and ResNet-based vehicle recognition. To emulate diverse traffic conditions as well as make the experimental results reproducible, we pre-load a vehicle RGB video dataset (i.e., BrnoCompSpeed [40]) and a vehicle thermal dataset (i.e., Teledyne FLIR [11]) on the edge nodes of the smart lampposts. BronCompSpeed contains 21 full-HD videos with 20,865 vehicles, and Teledyne FLOR thermal dataset contains 9,711 thermal images of 15 categories.

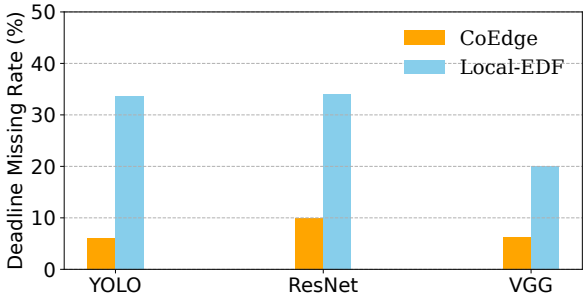
We choose a total of six smart lampposts on the representative road sections to conduct the experiments. Specifically, the lamppost nodes #1, 4, 6 are selected to execute the real-time traffic monitoring task. Since the detection range of the camera is 100 meters, while the distance between the two adjacent lampposts varies from 20 $\sim$ 60 m, we only run a single YOLO task on each of those nodes. The interval (i.e., task deadline) at which a new job is released is 1.3s. Under this setting, the detection results are obtained before the vehicle leaves the field of view of the lamppost, which avoids the accumulation of storage/compute load posed by incoming traffic. The lamppost nodes #1, 2, 3 are selected to perform pedestrian recognition since those nodes are close to a dining hall, which has a large flow of people traffic. Since the detection range of each thermal camera is around 20 $\sim$ 30 m, we run 2 pedestrian recognition tasks on each of those nodes with a task interval of 0.7s. Similarly, we select the lampposts near a parking lot for vehicle recognition. In summary, there is a total of 11 real-time DL tasks running on six lampposts. Table. 3 presents the setting of each application and the source edge nodes, i.e., the nodes on which the application data is

Table 3: Application settings and distribution.

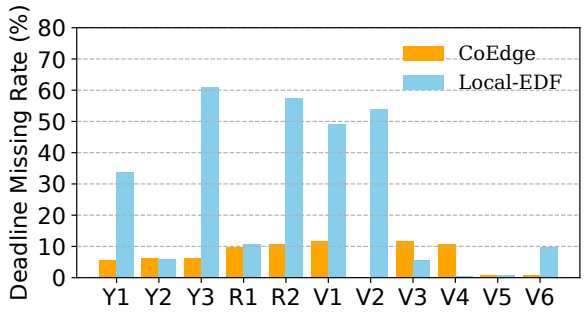
Application	Source Node	Model	Data	Task Interval
RT-Traffic-Monitor	1, 4, 6	YOLO	Camera	1.3s
Pedestrian-Rec	1, 2, 3	VGG	Thermal	0.7s
Vehicle-Rec	5, 6	ResNet	Camera	0.7s

Table 4: Task allocation results on smart lampposts.

Methods	Node1	Node2	Node3	Node4	Node5	Node 6
Local + EDF	Y1,V1,V2	V3,V4	V5,V6	Y2	R1	Y3,R2
CoEdge	Y1	V1,V2,V3	V4,V5,V6	Y2	R1,R2	Y3



(a) Deadline missing rate per model type

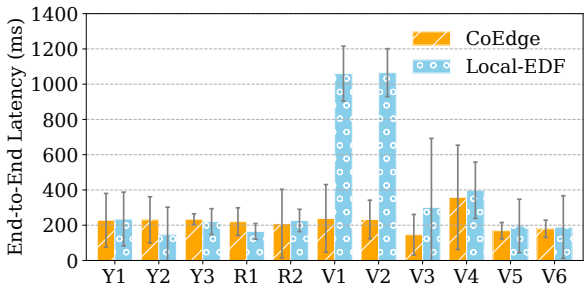


(b) Deadline missing rate per task

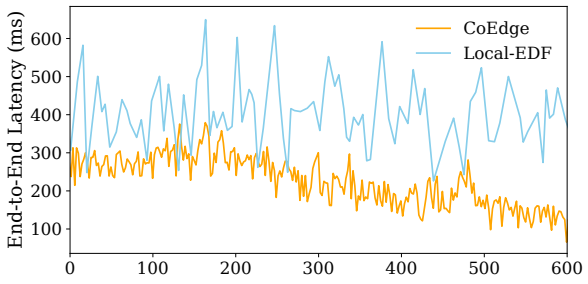
Figure 9: Deadline missing rates of different model types and tasks on smart lamppost testbed.

generated. We note that a task may be moved to another node for execution according to our task dispatching mechanism. In addition, we measure the communication bandwidth among different edge nodes (i.e., smart lampposts). We find that the bandwidths among smart lampposts vary between 50 ∼ 100 Mbps.

We evaluate the performance of CoEdge by comparing it against the local execution strategy with the Earliest Deadline First (EDF) scheduling policy (i.e., "Local-EDF"). EDF scheduling assigns the highest priority to the job with the closest absolute deadline. Local-EDF executes the DL tasks locally on their data source nodes and



(a) End-to-end latency per task



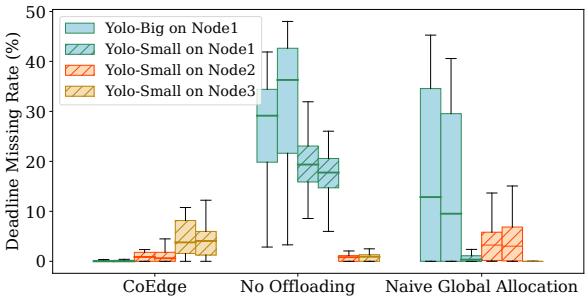
(b) End-to-end Latency per job

Figure 10: End-to-end latency of tasks and jobs on the smart lamppost testbed.

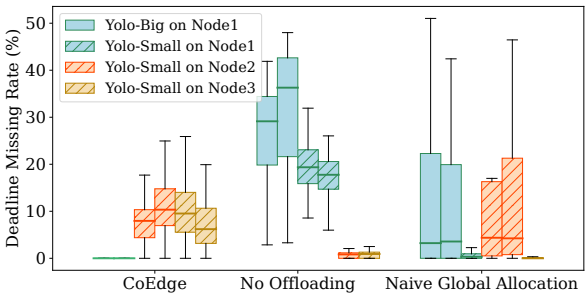
adopts EDF scheduling for concurrent DL task execution. Table. 4 shows the task allocation results by CoEdge and baseline Local-EDF. We observe that CoEdge successfully offloads the DL task from the heavy-load node to the light-load node (e.g., one ResNet-based DL task migrated from Node6 to Node5). CoEdge also successfully groups different DL tasks with the same DNN model into the same node (e.g., two VGG-based DL tasks migrated from Node1 to Node2).

We measure the deadline missing rate for each DNN model type and each DL task to evaluate the real-time performance. Fig. 9(a) shows the result for each model type. CoEdge consistently maintains a deadline missing rate 5.92%, 9.97% and 6.08% for YOLO, ResNet and VGG, respectively. In contrast, the baseline causes massive deadline misses, where 19.90% for VGG, 33.48% for YOLO, and 33.91% for ResNet. For example, CoEdge outperforms the baseline by relative 82.32%. Fig. 9(b) shows the deadline missing rate of each DL task. CoEdge maintains the deadline missing rate for each DL task below 10%. Specifically, it reduces the deadline missing rate by 54.67% for the Y3 task and 46.89% for the R2 task. The deadline missing rate of CoEdge is higher than that of the baseline for V3 and V4 tasks (but still under 10%). This is because CoEdge allocates the DL tasks from the other nodes to execute concurrently with the V3 and V4 tasks, which help meet the real-time requirements of the allocated tasks.

We further analyze the advantages of CoEdge by calculating the end-to-end latency per task and inference job. Fig. 10(a) shows the average latency for each DL task within a duration of 1,300s. We observe that CoEdge effectively reduces the average latency for each task (i.e., 220.11ms and 1065.32ms of CoEdge and Local-EDF for

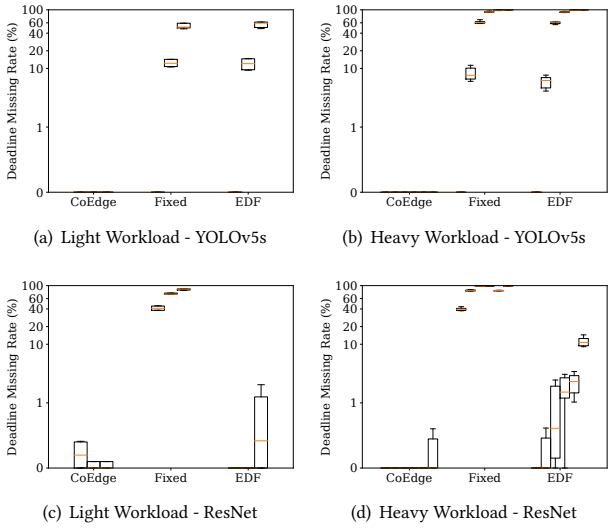


(a) High Bandwidth (1000Mbps)



(b) Low Bandwidth (100Mbps)

Figure 11: Real-time performance of CoEdge under different network bandwidths.



(a) Light Workload - YOLOv5s (b) Heavy Workload - YOLOv5s

(c) Light Workload - ResNet (d) Heavy Workload - ResNet

Figure 12: Real-time performance of batched DNN execution under different DNN workloads.

the V2 task, respectively). Fig. 10(b) shows the end-to-end latency of each job released by the V2 task. We observe that the end-to-end latency of each job of CoEdge is less than that of the baseline. This is because CoEdge allocates the VGG-based pedestrian recognition task from the busy node (i.e., Node 1) to the more spare node (i.e., Node 2), which allows more resources for the VGG-based task execution.

In conclusion, the evaluation in this section shows that CoEdge consistently maintains low deadline missing rates for smart traffic management task sets among multiple outdoor smart lampposts.

6.4 Impact of Network Bandwidth

We evaluate CoEdge under different network bandwidths on the indoor testbed. We compare CoEdge with two baselines, i.e., no offloading and a naive global allocation. To be specific, *no offloading* refers to the local execution of DL tasks with our batched DNN execution mechanism. The naive global allocation strategy allocates the DL tasks according to the real-time performance of each task. This baseline differs from CoEdge only in the global allocation policy, i.e., it allocates the tasks to different nodes only based on their deadline missing rates.

We test CoEdge under the bandwidth of 100 Mbps and 1,000 Mbps through the local area network, which are roughly the upper bounds of 4G [43] and the user experience data rates of 6G [41], respectively. To evaluate the fine-grained performance of CoEdge, we deploy tasks #1 and #2 in Table 1, which are two kinds of YOLOv5s with the input sizes of 640×640 and 160×160, respectively. In these experiments, we deploy two larger models ("YOLO-Big") and two smaller models ("YOLO-Small") on node 1, two YOLO-Small models on node 2 and node 3, respectively. The deadline of YOLO-Big is set to 1300ms, and the deadline of YOLO-Small is set to 600ms.

Fig. 11(a) and Fig. 11(b) show that CoEdge can effectively improve the real-time performance of DL tasks with different levels of bandwidths. Specifically, as shown in Fig. 11(a), CoEdge achieves 1.56% deadline missing rate at average, while 17.38% for without offloading and 4.83% for naive global allocation. For the low bandwidth, CoEdge can reduce the deadline missing rate by 11.71% compared with the baseline without workload offloading (i.e., No Offloading) under high bandwidth. The results show that CoEdge can achieve fewer deadline missing rates than the two baselines under both high and low bandwidths.

6.5 Performance of Batched DNN Execution

We evaluate batched DNN execution (c.f., Section 4.3) under different workloads induced by various settings of task numbers and model types. Specifically, we evaluate low and high workloads of task sets containing three and six typical DNN inference tasks, respectively. We test two models, i.e., ResNet and YOLO, under the two workloads. The deadline for each task is set to be around four times the measured average inference time of a single DL task. We use two baselines for this approach: Fixed-priority scheduling ("Fixed"), which assigns fixed priorities to tasks and does not change priorities over time; and Earliest Deadline First ("EDF") scheduling.

Fig. 12 shows the real-time performance of batched DNN execution in CoEdge under different DNN workloads. We record the deadline missing rate every 100 seconds and show their distribution in the figure. We also compute the median value of deadline missing rates, which are shown as orange lines. Fig. 12(a) shows that CoEdge achieves 0% deadline missing rate, where the average deadline missing rates are 21.03% for EDF and 23.75% for Fixed. As

shown in Fig. 12(c), CoEdge achieves more inferences for ResNet tasks in time than Fixed and EDF. Specifically, CoEdge can reduce the deadline missing rate by 40.86% for ResNet-1 and 71.66% for ResNet-2 (calculated by median difference) compared with Fixed. For heavy workload of YOLO, as shown in Fig. 12(b), CoEdge still can execute all the DL tasks in time (deadline missing rate is close to 0), while Fixed can only achieve 59.79% on average, and 59.64% for EDF. Moreover, for the heavy workload of ResNet, CoEdge can execute most DL tasks in time, while Fixed achieves 83.29% average deadline missing rate. In conclusion, CoEdge performs better than most other methods under both light and heavy workloads.

6.6 Overhead of GPU-Aware Concurrent DL Containerization

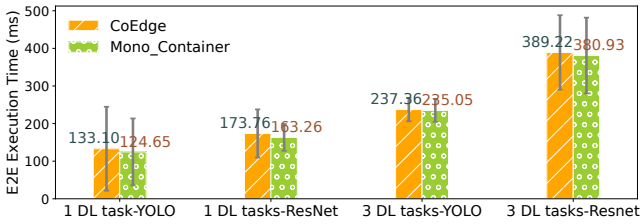


Figure 13: Impact of GPU-aware concurrent DL containerization.

The containerization of concurrent DL tasks in Section 4.4 isolates the execution of the runtime environment and enabling the simultaneous execution of DL tasks. However, containerization can lead to extra communication overheads among different containers. We evaluate this overhead by measuring the average end-to-end execution time of each DL task. We implement a baseline approach for evaluation, i.e. monotonic containerization ("Mono Container"). It encapsulates all DL tasks in a single container, which achieves simultaneous execution of DL tasks in a single monolithic runtime environment.

We evaluate CoEdge and the baseline with four task sets of a single ResNet, three ResNets, a single YOLO, and three YOLOs, respectively. Fig. 13 shows the average end-to-end execution time of four task sets from CoEdge and the baseline.

The results show that CoEdge only incurs a mere 8.45 ms extra execution time of a single YOLO model compared to the baseline. When executing three YOLO models, CoEdge only incurs 2.31 ms extra time compared to the baseline. For the task of ResNet, CoEdge incurs 10.5 ms and 8.29 ms extra communication time among containers for a single and 3 ResNet tasks compared to the baseline. The extra communication time for three DL tasks is less than that for a single DL task because the overhead is averaged. Hence, CoEdge achieves efficient GPU-aware concurrent DL containerization with negligible overheads, while offering isolated runtime environments for different DL tasks.

7 DISCUSSION

Application Scenarios of CoEdge. Although we evaluate CoEdge in a smart city scenario, CoEdge can also be applied to other applications based on the collaborative multi-edge architecture. For instance, CoEdge can support a wide range of edge applications like smart buildings, factories, and ports. In a smart port scenario, several types of edge nodes like smart equipment, lifts and trucks are present, while each edge node hosts various sensors, such as cameras and LiDARs. These nodes collaboratively provide services (i.e., DL tasks) like automatic identification of road trucks and motion tracking. To achieve efficient edge-based collaboration among smart equipment, CoEdge is capable of allocating DL tasks to each smart equipment using the global task dispatcher. Additionally, based on the local executor, CoEdge can support efficient concurrent execution of DL tasks on each edge node of the port.

Integration with Edge-Cloud Offloading. CoEdge can be integrated with existing edge-cloud offloading approaches, in which partial DL workloads (i.e., several model layers) are offloaded to a powerful cloud. In a typical existing offloading approach, an edge node may still suffer from resource contention since concurrent partial model inference workloads remain on the edge. In such a case, we may consider sharing the models for batch processing, since the same model with different offloading policies can share the same partial model inference. However, the processing time of the tasks sharing the same model can vary due to different offloading policies. Hence, when estimating batch processing time in the local executor (see Section 4.3), we need to account for the model offloading policy of each task.

Scalability to Large-Scale Applications. Although CoEdge is evaluated on a six-node lamppost testbed, the same design can be scaled up to a larger system. This is due to the fact that CoEdge nodes are only responsible for processing the data from nearby sensors, and are more likely to collaborate with their adjacent nodes, irrespective of the system scale. In the presence of more heterogeneous model types, CoEdge needs to conduct urgent task selection with more models on the local executor, and calculate the similarity with more tasks on the global dispatcher. However, the complexity of both algorithms remains the same at $O(n)$, where n is the number of tasks for the global dispatcher (see Section 4.2.2) and the number of model types for the local executor (see Section 4.3).

8 CONCLUSION

In this paper, we present a novel cooperative edge system CoEdge, which supports concurrent data/compute-intensive deep learning models for distributed real-time applications such as city-scale traffic monitoring and autonomous driving. CoEdge contains a hierarchical DL task scheduling framework that implements global task dispatching and batched DNN execution. Besides, CoEdge designs a GPU-aware concurrent DL containerization to support isolated execution environments for concurrent DNN execution on the edge platforms. We implement and evaluate CoEdge on a self-deployed smart lamppost testbed on a university campus. Our evaluations show that CoEdge achieves up to 82.32% reduction on deadline missing rate compared to baselines.

ACKNOWLEDGMENTS

The work described in this article was partially supported by the Innovation and Technology Commission of Hong Kong under Grant No. GHP/126/19SZ, and Research Grants Council (RGC)-General Research Fund under Grant No. 14211121.

REFERENCES

[1] Advantech. 2022. MIC-720AI - AI Inference System based on NVIDIA® Jetson Tegra X2. https://www.advantech.com/en-eu/products/9140b94e-bcfa-4aa4-8df2-1145026ad613/mic-720ai/mod_19d7f198-a3f3-4975-ac87-e8facd1045b3.

[2] Mohammed AA Al-qaness, Aaqif Afzaal Abbasi, Hong Fan, Rehab Ali Ibrahim, Saeed H Alsamhi, and Ammar Hawbani. 2021. An improved YOLO-based road traffic monitoring system. *Computing* 103, 2 (2021), 211–230.

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. Onnx: Open neural network exchange. *GitHub repository* (2019), 54. Online; accessed 4-March-2023.

[4] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. 2019. Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors* 19, 9 (2019), 2048.

[5] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. 2018. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 107–118.

[6] Baotong Chen, Jiafu Wan, Lei Shu, Peng Li, Mithun Mukherjee, and Boxing Yin. 2017. Smart factory of industry 4.0: Key technologies, application case, and challenges. *Ieee Access* 6 (2017), 6505–6519.

[7] Long Chen, Jigang Wu, Xin Long, and Zikai Zhang. 2017. ENGINE: Cost effective offloading in mobile edge computing with fog-cloud cooperation. *arXiv preprint arXiv:1711.01683* (2017).

[8] OpenFog Consortium et al. 1934. IEEE standard for adoption of OpenFog reference architecture for fog computing. *IEEE Std* 2018, 2018 (1934), 1–176.

[9] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 13733–13742.

[10] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 115–127. https://doi.org/10.1145/3241539.3241559

[11] Teledyne FLIR. 2022. FREE Teledyne FLIR Thermal Dataset for Algorithm Training. https://www.flir.asia/oem/adas/adas-dataset-form/.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Yuze He, Li Ma, Zhehao Jiang, Yi Tang, and Guoliang Xing. 2021. VI-Eye: Semantic-Based 3D Point Cloud Registration for Infrastructure-Assisted Autonomous Driving. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (New Orleans, Louisiana) *(MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 573–586. https://doi.org/10.1145/3447993.3483276

[14] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, and Yonghye Kwon. 2022. ultralytics/yolov5: V6. 1-TensorRT TensorFlow edge TPU and OpenVINO export and inference. *Zenodo* 2 (2022), 2.

[15] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[16] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[17] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. 2020. SPINN: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*. 1–15.

[18] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[19] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2022. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 91–105.

[20] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. 2021. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 1–14.

[21] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[22] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proc. IEEE* 107, 8 (2019), 1697–1716. https://doi.org/10.1109/JPROC.2019.2915983

[23] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* 7, 66 (2022), eabm6074. https://doi.org/10.1126/scirobotics.abm6074

[24] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. 2017. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358. https://doi.org/10.1109/COMST.2017.2745201

[25] Christian Meurisch Max Mühlhäuser. 2020. Street lamps as a platform. https://cacm.acm.org/magazines/2020/6/245163-street-lamps-as-a-platform/abstract

[26] Lifan Mei, Runchen Hu, Houwei Cao, Yong Liu, Zifa Han, Feng Li, and Jin Li. 2019. Realtime mobile bandwidth prediction using lstm neural network. In *Passive and Active Measurement: 20th International Conference, PAM 2019, Puerto Varas, Chile, March 27–29, 2019, Proceedings 20*. Springer, 34–47.

[27] Jiaying Meng, Haisheng Tan, Xiang-Yang Li, Zhenhua Han, and Bojie Li. 2019. Online deadline-aware task dispatching and scheduling in edge computing. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2019), 1270–1286.

[28] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. 2019. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2287–2295.

[29] Roberto Morabito. 2017. Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation. *IEEE Access* 5 (2017), 8835–8850. https://doi.org/10.1109/ACCESS.2017.2704444

[30] Zhaolong Ning, Peiran Dong, Xiangjie Kong, and Feng Xia. 2018. A cooperative partial computation offloading scheme for mobile edge computing enabled Internet of Things. *IEEE Internet of Things Journal* 6, 3 (2018), 4804–4814.

[31] NVIDIA. 2022. Nvidia TENSORRT. https://developer.nvidia.com/tensorrt.

[32] NVIDIA. 2023. Jetson TX2 Module. https://developer.nvidia.com/embedded/jetson-tx2.

[33] Jisun Oh, Seoyoung Kim, and Yoonhee Kim. 2018. Toward an adaptive fair GPU sharing scheme in container-based clusters. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 79–85.

[34] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. 2020. Toward Lighter Containers for the Edge.. In *HotEdge*.

[35] Lihua Ruan, Maluge Pubuduni Imali Dias, and Elaine Wong. 2019. Machine learning-based bandwidth prediction for low-latency H2M applications. *IEEE Internet of Things Journal* 6, 2 (2019), 3743–3752.

[36] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. 2022. VIPS: Real-Time Perception Fusion for Infrastructure-Assisted Autonomous Driving. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (Sydney, NSW, Australia) *(MobiCom '22)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/3495243.3560539

[37] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[38] Zhan Shi, Yongping Xie, Wei Xue, Yong Chen, Liuliu Fu, and Xiaobo Xu. 2020. Smart factory in Industry 4.0. *Systems Research and Behavioral Science* 37, 4 (2020), 607–617.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[40] Jakub Sochor, Roman Juránek, Jakub Špaňhel, Lukáš Maršík, Adam Širokỳ, Adam Herout, and Pavel Zemčík. 2018. Comprehensive data set for automatic single camera visual speed measurement. *IEEE Transactions on Intelligent Transportation Systems* 20, 5 (2018), 1633–1643.

[41] Petroc Taylor. 2022. User experience data rates of 4G, 5G and 6G technology. https://www.statista.com/statistics/1183674/mobile-broadband-user-data-rates/. Accessed: 2022.

[42] Yanan Wang, Nicolas Coudray, Yun Zhao, Fuyi Li, Changyuan Hu, Yao-Zhong Zhang, Seiya Imoto, Aristotelis Tsirigos, Geoffrey I Webb, Roger J Daly, et al. 2021. HEAL: an automated deep learning framework for cancer histopathology image analysis. *Bioinformatics* 37, 22 (2021), 4291–4295.

[43] Wikipedia. 2023. 4G. https://en.wikipedia.org/wiki/4G. Accessed: 2023.

[44] Hongyue Wu, Shuiguang Deng, Wei Li, Samee U Khan, Jianwei Yin, and Albert Y Zomaya. 2018. Request dispatching for minimizing service response time in edge cloud systems. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[45] Xinglong Wu, Shangbin Chen, Jin Huang, Anan Li, Rong Xiao, and Xinwu Cui. 2020. DDeep3M: Docker-powered deep learning for biomedical image segmentation. *Journal of Neuroscience Methods* 342 (2020), 108804.

[46] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 392–405.

[47] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. 2018. Extend cloud to edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 373–377.

[48] Dianlei Xu, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. 2021. Edge Intelligence: Empowering Intelligence to the Edge of Network. *Proc. IEEE* 109, 11 (2021), 1778–1837. https://doi.org/10.1109/JPROC.

2021.3119950

[49] Shenghao Yang and Raymond W Yeung. 2017. BATS Codes: Theory and practice. *Synthesis Lectures on Communication Networks* 10, 2 (2017), 1–226.

[50] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. 2018. FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices *(SenSys '18)*. Association for Computing Machinery, New York, NY, USA, 278–291. https://doi.org/10.1145/3274783.3274840

[51] S. Yi, Z. Hao, Z. Qin, and Q. Li. 2015. Fog Computing: Platform and Applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 73–78. https://doi.org/10.1109/HotWeb.2015.22

[52] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-Meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[53] Shanshan Zhang, Rodrigo Benenson, Mohamed Omran, Jan Hosang, and Bernt Schiele. 2016. How far are we from solving pedestrian detection?. In *Proceedings of the iEEE conference on computer vision and pattern recognition*. 1259–1267.

[54] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. 2018. ECRT: An edge computing system for real-time image-based object tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 394–395.

[55] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. 2021. EdgeML: An AutoML framework for real-time deep learning on the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 133–144.