EdgeBatch: Towards AI-empowered Optimal Task Batching in Intelligent Edge Systems

Daniel (Yue) Zhang, Nathan Vance, Yang Zhang, Md Tahmid Rashid, Dong Wang
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA
{yzhang40, nvance1, yzhang42, mrashid, dwang5}@nd.edu

***Abstract*—Modern Internet of Things (IoT) systems are increasingly leveraging deep neural networks (DNNs) with the goal of enabling intelligence at the edge of the network. While applying DNNs can greatly improve the accuracy of autonomous decisions and inferences, a significant challenge is that DNNs are traditionally designed and developed for advanced hardware (e.g., GPU clusters) and can not easily meet the real time requirements when deployed in a resource-constrained edge computing environment. While many systems have been proposed to facilitate deep learning at the edge, a key limitation lies in the underutilization of the parallelizable GPU resources of edge nodes (e.g., IoT devices). In this paper, we propose EdgeBatch, a collaborative intelligent edge computing framework that minimizes the delay and energy consumption of executing DNN tasks at the edge by sharing idle GPU resources among privately owned IoT devices. EdgeBatch develops 1) a stochastic task batching mechanism that identifies the optimal batching strategy for the GPUs on IoT devices given uncertain task arrival times, and 2) a dynamic task offloading scheme that coordinates the collaboration among edge nodes to optimize the utilization of idle GPU resources in the system. We implemented EdgeBatch on a real-world edge computing testbed that consists of heterogeneous IoT devices (Jetson TX2, TX1, TK1, and Raspberry Pi3s). The results show that EdgeBatch achieved significant performance gains in terms of both the end-to-end delay and energy savings compared to the state-of-the-art baselines.**

I. Introduction

The rise of Internet of Things (IoT) and Artificial Intelligence (AI) leads to the emergence of Intelligent Edge Systems (IES) that run AI models on the IoT devices at the edge of the network [1]. A core IES technique is deep learning (deep neural networks (DNN) in particular). Unlike traditional deep learning solutions that offload the computationally intensive inference tasks from the IoT devices to the cloud, IES directly execute those tasks at the edge and provide several key advantages (e.g., reduced bandwidth cost, improved responsiveness, and better privacy protection) to the system [2]. Examples of IES applications include mobile augmented reality (AR) on smart devices [3], disaster damage assessment using citizen-owned cameras [4], and traffic monitoring using vehicle dashboard cameras [5].

Pushing intelligence to the IoT devices in IES is a major challenge because DNNs were originally designed for advanced hardware (e.g., GPU clusters) and are not suitable for resource constrained IoT devices deployed at the edge of the network [6]–[9]. Moreover, running DNN algorithms often incurs a high energy cost, which may rapidly drain the batteries of IoT devices [10], [11]. To address this challenge, many software and hardware based approaches have been developed. Mainstream techniques include 1) neural network compression that reduces the size and computational complexity of the DNNs [12], [13], 2) dedicated equipment with on-board hardware such as AI Chips and powerful GPUs that are specialized for DNN tasks (e.g., AWS DeepLens [14] and Nvidia EGX [15]), and 3) innovative software accelerators that increase the energy efficiency and speed for DNN execution (e.g., DeepX [16] and NVDLA [17]).

A key knowledge gap of the above solutions lies in the fact that existing approaches focus on facilitating DNNs on a *single IoT device* and ignore the opportunities to optimize the performance of DNNs collectively in an IES environment that consists of a diverse set of heterogeneous IoT devices connected via network. In contrast, this paper develops a novel resource management approach where IoT devices can offload deep learning tasks to each other and finish them collaboratively. For example, consider an IES application where drivers use their IoT devices (e.g., smartphones, dashboard cameras, unmanned aerial vehicles (UAVs)) to collaboratively detect the plate number of a suspect's vehicle using deep neural network object detection algorithms [5]. In our solution, lower-end devices (e.g., a dashboard camera) can offload complex object detection tasks to high-end devices (e.g., a UAV with a GPU on board). Such collaboration in IES allows IoT devices to fully explore the available computing power at the edge to execute DNN tasks.

A few recent efforts have been made to facilitate the collaboration of privately owned IoT devices through innovative task allocation [18], [19] and incentive design [20], [21]. However, they only focus on CPU-intensive tasks and ignore the unique execution model of GPU-intensive DNN tasks. The GPU execution model features *data parallelism*, which allows multiple DNN tasks to be processed together (referred to as "batching") and reduces the average execution time for each task [22]. In this paper, we focus on a novel *optimal task batching* problem in collaborative IES, where the goal is to identify the optimal batch size (i.e., the number of tasks to be processed in parallel) when the DNN tasks are processed on the GPUs of IoT devices. While batching problems have been studied extensively in traditional real-time systems [23], [24], the optimal batching of DNN tasks in IES raises several new technical challenges.

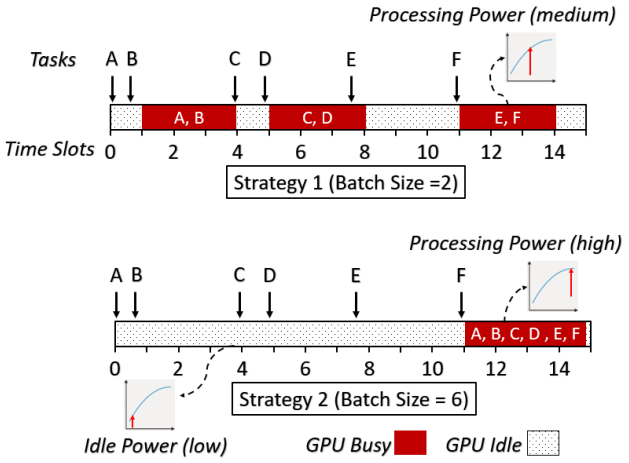


Figure 1: Delay and Energy Trade-off of Batch Size

Complex delay and energy trade-off: the first challenge lies in the complex trade-off of delay and energy that is directly affected by the task batching strategy. In a single node scenario, batching can significantly save processing time of DNN tasks through parallel computing but causes extra power consumption. For example, on a typical edge computing device such as Nvidia TX2, increasing the batch size from 1 to 5 leads to 18% decrease in delay but 3.4% increase in average power consumption. Such trade-off is more complex in a distributed IES where the delay not only includes the processing time of the DNN tasks, but the unpredictable task waiting time as well as data offloading overhead. We illustrate this trade-off through an example in Figure 1. Batching Strategy 1 (top) uses a batch size of 2 tasks, and Batching Strategy 2 (bottom) uses a batch size of 6 tasks. We first discuss the trade-off between delay and batch size. We observe that a larger batch size will reduce the average processing time of a task but increase the total task waiting time (e.g., in Strategy 2, task A will have to wait until task F arrives before being processed). In contrast, a smaller batch size will lead to less waiting time but yield a higher execution time for each task due to the under-utilization of the GPU [22]. A similar trade-off is observed on the energy aspect as well. For example, a larger batch size would lead to a longer idle period where GPU is in a low power state. However, the processing power of GPU will become higher due to its increased utilization caused by the larger batch of data being processed [25]. It is therefore a challenging task to identify the optimal batch size that can achieve a desirable balance between the delay and energy requirements of DNN applications in IES.

Uncertain Task Arrival: The second challenge lies in the uncertainty of the task arrival time in IES. Existing work in task batching either assumes the task release time or task period is known *a priori* [24] or assumes the task arrival time follows a certain distribution or pattern [26]. In contrast, our model assumes the task release time is unpredictable and the tasks can arrive at an IoT device at random times, thus making the task batching problem significantly more challenging. The rationale behind our assumption is twofold: 1) the DNN tasks are released whenever new sensing measurements are collected, which is unknown in many IES applications; and 2) the networking environment (e.g., the queue size, available bandwidth, router status) is dynamically changing and the data transmission time cannot be precisely estimated in IES. Consider the plate detection application where drivers can take a picture of a car at any moment. When the image processing tasks are offloaded over the network, the transmission time depends on the signal strength and the network traffic, both of which are hard to predict.

Asymmetric Information: the third challenge lies in the fact that both IoT devices and the application server lack the global knowledge of system status. In a distributed system like IES, locally optimal task batching decisions may not necessarily be globally optimal. Consider an IES scenario that incorporates heterogeneous IoT devices. These IoT devices can have very different architecture, energy profile, and computing capabilities. In principle, it is ideal to allocate more tasks to devices that have abundant GPU resources (e.g., a nearby UAV that is not actively using its GPU) so that the devices can efficiently perform task batching without an excessive waiting time. However, such a control mechanism requires the global knowledge of system information (e.g., the hardware specifications, the GPU utilization, the remaining battery of each device) and centralized control of all devices in the system. Unfortunately, the IoT devices in IES are often owned by individuals who may not be willing to share the status information of their devices with the application or other users due to various concerns (e.g., privacy, energy, bandwidth) [27]. Therefore, an IoT device in IES has full information about its own status but limited information about others. This asymmetric information trait of IES makes the optimal task batching decisions over heterogeneous IoT devices a challenging task to accomplish.

In this paper, we develop a new collaborative IES framework called EdgeBatch to jointly address the above challenges. In particular, to address the first two challenges, EdgeBatch develops a new task batching solution that can identify the optimal batch size for GPU-intensive DNN tasks to optimize the trade-off between the end-to-end delay of tasks and energy consumption for IoT devices. To address the third challenge, EdgeBatch develops a novel supply chain-based task offloading scheme that can effectively moderate the collaboration among IoT devices in real-time to facilitate the batching decision process without requiring the private status information of the IoT devices. To the best of our knowledge, EdgeBatch is the first DNN task batching solution with uncertain task arrival time and information asymmetry for collaborative IES applications. We implemented a system prototype of EdgeBatch on a real-world collaborative edge testbed that consists of Nvidia Jetson TX2, TX1 and TK1 boards, and Raspberry Pis. EdgeBatch was evaluated using a real-world DNN application: *Crowd Plate Detection*. We compared EdgeBatch with state-of-the-art baselines in real-time and edge computing systems. The results show that our scheme achieves a significant performance gain in terms of

both the end-to-end delay and energy savings.

II. Related Work

A. AI at The Edge

AI at the edge is a growing trend in both industry and academic research. The real-time response requirement of IoT applications, in conjunction with data privacy and network connectivity issues, call for *intelligent* edge devices that are able to support delay-sensitive computation for deep learning on-site [1], [2]. Many solutions have been developed to promote deep learning using IoT devices. One common technique is "neural network compression" which can significantly reduce the complexity of the neural network so that it can be run on resource constrained IoT devices efficiently. For example, Yao *et al.* developed DeepIoT, a neural network compression framework that reduces the number of parameters by over 90%, resulting in a significant reduction in execution time and energy cost of running deep neural networks at the edge [12]. Han *et al.* developed Deep Compression - a pipeline of pruning, quantization, and Huffman coding techniques to reduce both the storage and energy consumption of deep neural networks [13]. Many hardware solutions have also been proposed. For example, AI-enabled chips and dedicated hardware have been developed and integrated into video cameras, handheld devices, and vehicles to allow edge devices to run deep learning tasks efficiently. Typical hardware include EdgeBox from Microsoft [28], DeepLens from AWS [14], and TPU from Google [29]. In addition, some hardware accelerators have been proposed to further speedup the computation at the edge with a low energy cost [8], [30]. There are also several recent software-hardware co-design approaches that extract both the data and control flow parallelism. In particular, they design hardware accelerators for some parts of DNN codes and assign other parts to CPUs or GPUs [31], [32]. In this work, we propose an alternative approach to facilitate AI at the edge by introducing a new task batching scheme in collaborative IES.

B. Task Offloading in Edge Computing

One of the key problems studied in resource management in edge computing is *task offloading* (sometimes referred to as computation offloading) which is the transfer of resource intensive computational tasks to an external server/device [33]. Task offloading is particularly important for resource constrained devices that cannot process complex tasks on their own, especially under a strict timing requirement [34]. Existing task offloading systems are mostly top-down approaches which assume that a centralized decision maker (often an algorithm running on the back-end server) makes global offloading decisions with the assumption that edge devices are fully controlled and cooperative [27], [35]. For example, Ning *et al.* formulated a Mixed Integer Linear Programming based approach to offload computational tasks from IoT devices to nearby servers that minimize the end-to-end latency [36]. Wang *et. al* developed a dynamic service migration scheme using a Markov decision process that dynamically adjusts the task offloading strategy of the mobile devices when the locations of device owners change over time [37]. These top-down approaches cannot address our problem due to the asymmetric information challenge where each device's energy profile and dynamic status are hidden from the applications [27]. To address the limitations of centralized task offloading schemes, decentralized decision-making schemes have been developed in edge computing systems, where decisions are made autonomously by IoT devices. For example, Zhang *et al.* developed CoGTA, an edge computing system that allows non-cooperative and heterogeneous devices to trade tasks and claim rewards [20]. Jin *et al.* proposed a game-theoretic decentralized task offloading protocol for a dynamic and uncertain environment [38]. However, the above solutions do not utilize data parallelization of GPUs for computation intensive deep learning tasks. In contrast, we develop a new collaborative IES framework that combines an optimal task batching scheme with a new task offloading scheme to fully leverage the idle GPU resources at the edge.

C. Parallel Computing and Task Batching

Parallel computing allows a machine to process several jobs simultaneously, thus significantly increasing the efficiency and utilization of the system [39]. A standard approach to achieve parallelization is task batching where a scheduler assigns multiple tasks to the processing unit and process them in parallel [23]. For example, Wang *et al.* developed a real-time batching scheme that finds energy-optimal batching periods for asynchronous tasks on heterogeneous sensor nodes to minimize energy consumption with end-to-end deadline constraints [24]. He *et al.* introduced a Batched Stream Processing (BSP) model that identifies the optimal data size to be processed for large-scale data streams [40]. A set of batching mechanisms have been developed for content delivery applications such as music streaming and video sharing [41], [42]. These models mainly focus on CPU-intensive or I/O tasks and are not applicable to the GPU-intensive DNN tasks that we study in this paper. A few GPU batching techniques have been developed to enable data parallelism for GPU tasks [43], [44] by parallelizing large matrix operations (which are key operations in deep learning tasks). However, these solutions are designed for a single GPU rather a networked edge computing system that we study in this work. Moreover, these models assume prior knowledge of either task arrival time or task period. We found the challenging problem of designing an optimal task batching strategy for GPU intensive tasks with stochastic task arrival time in a collaborative and distributed edge computing system has not been addressed in previous literature.

III. Problem Formulation

In this section, we formally define the models and objectives of the EdgeBatch framework.

A. System Models and Assumptions

Figure 2 illustrates an example of a DNN application of IES called *crowd plate detection*, where a set of private vehicles collaboratively track down suspects of AMBER alerts [5]. In

this application, IoT devices (e.g., vehicles equipped with dash cameras and smart devices owned by citizens) form a city-wide video surveillance network that tracks moving vehicles using the automatic license plate recognition (ALPR) technique [45].

In an IES application, the IoT devices not only collect sensor data but also perform deep learning tasks to process the data at the edge. In the plate detection example, the smartphones can not only capture images of the suspect's car, but also perform deep learning algorithms to detect the plate number in the captured images.

In addition to the IoT devices, a set of edge servers (e.g., cloudlets, smart routers, or gateways) are deployed by the application to provide additional data storage and computation power in locations of proximity to the IoT devices. In the above plate detection application, the application deploys Road-side-units (RSUs) on streets of interest as edge servers to coordinate the nearby IoT devices (e.g., the vehicles currently located on the street). These edge servers provide local data processing capabilities to reduce the end-to-end latency and offer a generic communication interface between heterogeneous IoT devices in the system [3], [10]. We refer to the sensing and computational resources at the edge (i.e., IoT devices and edge servers) as *edge nodes* $EN = \{E_1, E_2, ..., E_N\}$, where N is total number of edge nodes in the system.

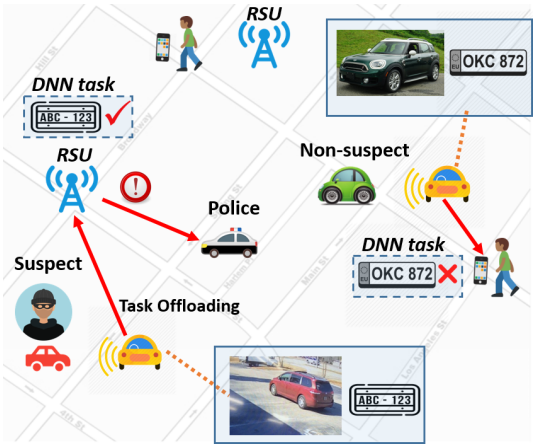


Figure 2: An Example of Plate Recognition Application

A key enabling technology in collaborative IES is *task offloading*, where an IoT device can choose to offload the data processing tasks to any device/service in the collaborative IES. Due to the dynamic nature of the IES system (e.g., the status of the computing nodes, the task pool, and the network environment can change over time), the task offloading strategies also need to be dynamic. In our model, we define the IES application as a time-slotted system with a total of T time slots. We use $t \in [1, T]$ to denote the t-th time slot.

We then leverage the terms in supply chain models in economics [46] to define three types of edge nodes *supplier*, *manufacturer*, and *consumer* in the context of task offloading.

DEFINITION 1. Supplier ($\mathcal{S}$): the edge node that collects the sensing data.

DEFINITION 2. Manufacturer ($\mathcal{M}$): the edge node that performs the DNN tasks to process the data.

DEFINITION 3. Consumer ($\mathcal{C}$): the edge node that receives the final result from the DNN tasks.

We use $E_i \in \mathcal{S}, E_i \in \mathcal{M}$ and $E_i \in \mathcal{C}$ to denote an edge node is a supplier, manufacturer, or consumer, respectively. Note that a given node can be of multiple types. For example, a node that is both supplier and manufacturer collects and processes the data locally, and we use $E_i \in \mathcal{S}$ & $E_i \in \mathcal{M}$ to represent such a situation. Take the plate detection application as an example. The suppliers are the vehicles that use dashboard cameras to take images of car plates. The manufacturers are the edge nodes (e.g., the RSUs) that process the images, and the consumer can be the device owned by the response team (e.g., the tablet in a police vehicle).

The goal of task offloading is to identify an optimal *dynamic supply chain graph* of the system as defined below.

DEFINITION 4. Dynamic Supply Chain Graph ($\mathbf{G}^t_{\text{edge}}$): a directed 3-partite graph $\mathbf{G}^t_{\text{edge}} = (\mathbf{V}^t_S, \mathbf{V}^t_M, \mathbf{v}^t_C, \mathbf{L}^t_{SM}, \mathbf{L}^t_{MC},)$ where vertex $V^t_i \in \mathbf{V}^t_S, \mathbf{V}^t_M, \mathbf{v}^t_C$, represents that the edge node E_i is a supplier, manufacturer, or consumer, respectively at time slot t. Link $E_i \in \mathcal{S} \rightarrow E_j \in \mathcal{M} \in \mathbf{L}^t_{SM}$ signifies that supplier E_i collects the data and offloads the DNN tasks to manufacturer E_j at time slot t. Link $E_i \in \mathcal{M} \rightarrow E_j \in \mathcal{C} \in \mathbf{L}^t_{MC}$ signifies that manufacturer E_i sends the final results to the consumer E_j at time slot t.

For example, a supply chain of $E_a \rightarrow E_b \rightarrow E_c$ represents that E_a first collects the data, then asks E_b to process the data, which then sends the final result to E_c. Similarly, a supply chain $E_a \rightarrow E_a \rightarrow E_b$ denotes that node E_a collects the image data and run the DNN task on-board, and then send the final result to node E_c. The superscript t for the supply chain graph $\mathbf{G}^t_{\text{edge}}$ and links (e.g., $\mathbf{L}^t_{SM}$) refers to the fact that the supply chain is dynamically changing, depending on the availability of edge devices, the status of the nodes, as well as the workloads in the system. We discuss in details on how such dynamic supply chains are formed in Section IV-B.

B. Task Model

This paper focuses on a representative category of deep learning applications at the edge: deep neural network based image analysis [47]. We define a *task* that converts the captured image data to the final analysis results by running a Convolutional Neural Network (CNN), a widely used neural network algorithm for image analysis [47]. A CNN task is commonly processed using deep learning frameworks (e.g., Tensorflow, Caffe) that run on a GPU.

The key research problem we study in this paper is task batching. We assume that the above CNN tasks can be run on a GPU in parallel. The number of tasks that are executing simultaneously on GPU is referred to as the batch size of an edge node:

DEFINITION 5. Task Batching Size $|\mathbf{B_{m,i}}|$: the number of CNN tasks to be processed simultaneously on the GPU of an edge node E_i at the m-th batch. Assuming the the IES application is finite, each edge node E_i processes the tasks in a total of $M(i)$ batches $B_{1,i}, B_{2,i}, ...B_{M(i),i}$. The size (i.e., number of tasks to be processed) of a batch $B_{m,i}$ is denoted as $|B_{m,i}|$.

IoT devices collect sensing data from the physical world (i.e., collect and report an image from the camera). A collected image triggers a corresponding CNN task to process it. We assume that the application generates a set of $K(t)$ tasks at the time slot t, $TK^t = \{\tau_1^t, \tau_2^t, ...\tau_{K(t)}^t\}$. Each task is associated with a 3-tuple $\tau_k^t = (\Gamma_k^{t(R)}, \Gamma_k^{t(A)}, \Delta)$. Here $\Gamma_k^{t(R)}$ is the task release time which is the time slot when the image has been collected. $\Gamma_k^{t(A)}$ is the task arrival time which is the time slot when the image reaches a manufacturer. If the manufacturer is the same node as the supplier, we set $\Gamma_k^{t(A)} = \Gamma_k^{t(R)}$. Δ is the deadline requirement that captures the user desired Quality of Service (QoS). In this work, we assume all tasks are homogeneous (i.e., same priority level, input type, and algorithm to execute) and set all tasks to have the same deadline. This task model is quite common in image detection applications using deep learning techniques [4], [48]. We discuss how EdgeBatch can handle heterogeneous task models in Section VII.

Note that in an IES application scenario, each task can be accomplished by either one or more devices. The extra communication delay will be incurred whenever a device offloads its data to another. Formally, an extra communication task will be generated to perform data offloading for every supply chain link, i.e., $E_i \in \mathcal{S} \rightarrow E_j \in \mathcal{M} \in \mathbf{L}_{SM}^t$ when $i \neq j$, and $E_i \in \mathcal{M} \rightarrow E_j \in \mathcal{C} \in \mathbf{L}_{MC}^t$ when $i \neq j$. In the scenario where the supplier is also the manufacturer (i.e., the image data is processed on the same node), or the manufacturer happens to be the consumer, no additional communication task is generated for data offloading.

C. Energy Model

Energy consumption is a major concern for battery constrained IoT devices. In this paper, the energy consumption of an edge node E_i is derived as: $e_i = \int_1^T (P_{\text{Comp},i}^t + P_{\text{Trans},i}^t)\, dt, 1 \leq i \leq N, 1 \leq t \leq T$, where P_{comp}^t is the power consumption for computation and P_{trans}^t is power for data transmission via wireless network for edge node E_i at time slot t. Note that P_{comp}^t can be highly dynamic and the relationship between power consumption and the workload can be non-linear and time-varying [49]. Therefore, we do not assume the application is able to precisely estimate the energy consumption on each edge node in the system. Instead, we assume each edge node can measure the power consumption on its own (e.g., through the built-in energy modules) and do not share such information with the application. We discuss the energy measurement in Section V.

D. Objectives

The QoS of the application is defined as the end-to-end delay (E2E delay) of each task:

DEFINITION 6. End-to-end Delay of A Task ($\mathcal{D}_{\mathbf{k}}^{\mathbf{t}}$): the total amount of time taken for a task to transform a unit of sensor measurement (i.e., an image) to be processed and sent to the consumer node. It includes the total computation time of the CNN to process the image in task τ_k^t, the overhead of the EdgeBatch modules, and the total communication overhead for additional data offloading process in τ_k^t.

Based on the definitions and models discussed above, we formally define the objective of EdgeBatch as follows. Our goal is to develop an optimal task batching scheme to minimize the total energy consumption of the edge nodes and the end-to-end (E2E) delay of the application simultaneously. Therefore, we formulate our problem as a multi-objective optimization problem that targets at finding the optimal task batching sizes $|B_{i,m}|, 1 \leq i \leq N, 1 \leq m \leq M(i)$ for each edge node that can:

$$\begin{aligned} &\text{minimize:} \quad e_i, \forall 1 \leq i \leq N \\ &\text{minimize:} \quad \sum_{t=1}^{T} \sum_{k=1}^{K(t)} \mathcal{D}_k^t, \forall 1 \leq k \leq K, 1 \leq t \leq T \\ &\text{given:} \quad \mathbf{G}_{\text{edge}}^t, 1 \leq t \leq T \end{aligned} \tag{1}$$

IV. The EdgeBatch Framework

In this section, we introduce our EdgeBatch framework to solve the problem formulated in the previous section. EdgeBatch consists of two sub-modules (Figure 3): 1) a local Stochastic Optimal Task Batching (SOTB) module that identifies the optimal batch size of CNN tasks to fully utilize the data parallelization of GPU resources on edge nodes; 2) a global Optimal Contracting with Asymmetric Information (OCAI) module that manages the supply chains of the edge nodes to further utilize the idle GPU resources in the edge. The two modules work interactively to minimize energy and delay of the system. Note that the task batching problem with a single objective (e.g., makespan minimization) has been proven to be NP-Hard in the strong sense [50], [51]. The problem becomes even more challenging due to the multi-objective formulation in our optimization problem in Equation (2) and the challenges presented in Introduction. Therefore, we found it impractical to obtain an efficient globally optimal solution for the optimization problem in Equation (2). Therefore, we break down our problem into two sub-problems (i.e., optimal batching solved by SOTB and the task offloading solved by OCAI) to make the problem tractable. The detailed discussion of the optimality of each sub-module is presented at the end of following subsections.

A. Stochastic Optimal Task Batching Module (SOTB)

The SOTB module is designed to decide the optimal task batch size in real-time to explore a desired trade-off between delay of tasks and energy consumption of the IoT devices.

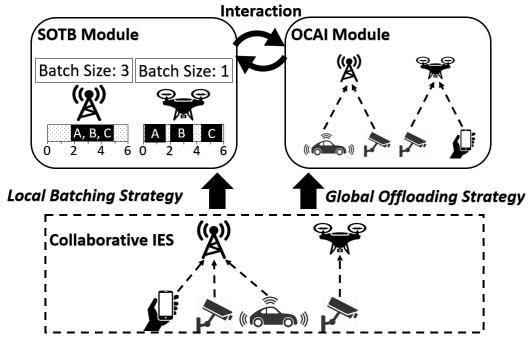


Figure 3: Overview of the EdgeBatch Framework

In a collaborative IES, a manufacturer node cannot precisely predict when a supplier would offload the sensing data to it. For example, a manufacturer has received 3 images from suppliers after the last processing batch. The manufacturer, without knowing the arrival time of the next image, needs to decide whether to wait for the 4^{th} image or process the 3 images immediately to avoid excessive waiting time. This problem shares the same intuition as the *bus waiting problem* in the transportation planning , where a driver needs to decide how long a bus needs to wait at a bus stop to balance 1) the average waiting time of the arriving passengers, and 2) the chance of delayed arrival to the destination caused by the wait [26]. However, our problem is much more complex than the bus waiting problem because we do not assume prior distribution of the task arrival time and we need to consider the energy trade-off caused by batching in the system.

We first formally define the mathematical model and key terms used in the SOTB module. For an edge node $E_i \in EN$, we assume it processes its tasks in a total of M batches, $\{B_1, B_2, ..., B_M\}$ where B_m denotes the m-th batch, $1 \leq m \leq M$. For ease of notation, we ignore the index for the edge node (i.e, the subscript i) in this subsection. Each batch B_m is associated with a 3-tuple: $B_m = (|B_m|, \Gamma^s_m, \Gamma^e_m)$, where $|B_m|$ is the batch size defined in Section III. Γ^s_m and Γ^e_m are the batch start and end time respectively. For example, $\Gamma^s_m = t_1$ and $\Gamma^e_m = t_2$ denote the B_m starts at time slot t_1 and ends at time slot t_2.

We further define two cost functions that are associated with the batch size below.

DEFINITION 7. Batch Delay Cost Function $\mathbf{f}(\cdot)$: the average processing delay for a specific batch size. In particular, $f(|B_m|)$ denotes the average processing delay of B_m.

DEFINITION 8. Batch Power Cost Function $\mathbf{g}(\cdot)$: the average computation power consumption for a specific batch size. In particular, $g(|B_m|)$ is the average power consumption of B_m.

We assume the two cost functions are non-linear and are known only by the edge node itself but unknown to other nodes/servers. Therefore, each edge device needs to compute $f(\cdot)$ and $g(\cdot)$ based on its unique computing power and energy profile. We elaborate the profiling of these cost functions for different types of edge nodes in Section VI. The batch size also affects the *batch holding time* defined below:

DEFINITION 9. Batch Holding Time $\mathbf{H_m}$: the time the edge node needs to wait it till processes the next batch B_m. It is formally calculated as: $H_m = \Gamma^s_m - \Gamma^e_{m-1}$, where a larger batch size would result in a longer holding time.

We further assume that each edge node has a *capacity* Θ which is defined as the maximum number of images that can be processed by the GPU on the device. When the edge node reaches its capacity, adding more images will not reduce the average task processing delay.

To model the trade-off between delay and energy, we define two loss functions: *delay loss* and *energy loss*.

DEFINITION 10. Delay Loss $\mathbf{W^{(D)}_m}$: the total delay for all tasks if performing batching B_m, including both processing delay and waiting delay.

DEFINITION 11. Energy Loss $\mathbf{W^{(E)}_m}$: the total energy costs of tasks if performing batching B_m, including the energy cost during both GPU idle and execution slots.

The delay loss $W^{(D)}_m$ depends on three factors: 1) the delay of the tasks that were left behind from the previous batch $\mathcal{L}_1$; 2) the processing time of the current batch $\mathcal{L}_2$; and 3) the total waiting time of the tasks arrived between the previous batch and the current batch $\mathcal{L}_3$. We have:

$$W^{(D)}_m = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3 \tag{2}$$

$\mathcal{L}_1$ can be derived as the number of left-over images from last batch times the batch holding time of the current batch B_m. Formally, we have

$$\mathcal{L}_1 = H_m \cdot L_{m-1} \tag{3}$$

where L_{m-1} denotes the left-over images from last batch to be processed at B_m. It is recursively defined as follows:

$$\begin{aligned} L_{m-1} = &\max[(\Gamma^s_{m-1} - \Gamma^e_{m-1}) \cdot R_{\Gamma^s_{m-1}, \Gamma^e_{m-1}} \\ &+ L_{m-2} - \Theta, 0] \end{aligned} \tag{4}$$

where $R_{\Gamma^s_{m-1}, \Gamma^e_{m-1}}$ is the average task arrival rate in $[\Gamma^s_{m-1}, \Gamma^e_{m-1}]$.

$\mathcal{L}_2$ is derived as the batch delay cost $f(\cdot)$ multiplied by the number of images to be processed. Formally, we calculate $\mathcal{L}_2$ as:

$$\begin{aligned} \mathcal{L}_2 &= f(|B_m|) \cdot |B_m| \\ &= \Big(\sum_{I \in B_m} f^1(I) + f^2(|B_m|) \Big) \cdot |B_m| \end{aligned} \tag{5}$$

where $I \in B_m$ denote an image I in the batch B_m. $f^1(I)$ is the preprocessing (e.g., encoding and resizing of the image) time of an image, which must be done before CNN algorithm runs on the GPU [47]. $f^2(|B_m|)$ is the actual execution time (parallelizable) on GPU given the batch size.

$\mathcal{L}_3$ is derived as the total amount of waiting time for images that arrive between the start of B_{m-1} and the end of B_m. Formally, we derive $\mathcal{L}_3$ as:

$$\mathcal{L}_3 = \sum_{I \in A_m} (\Gamma^s_m - \Gamma_I) \tag{6}$$

where A_m denotes the set of the images that arrive between the $(m-1)$-th and the m-th batch, and $\Gamma_I, \Gamma^e_{m-1} \leq \Gamma_I \leq \Gamma^e_m$ denotes the arrival time of an image I. Note that in this definition, $\mathcal{L}_3$ explicitly considers the the communication delay of transmitting the images in A_m.

We illustrate an example of delay cost breakdown in Figure 4. In this example, the previous batch B_{m-1} has a left-over Task D which cannot be processed by the GPU. Therefore, it suffers both holding delay ($\mathcal{L}_1$) and processing delay $\mathcal{L}_2$. Task E arrives between the two batches and suffers from waiting for B_m to start ($\mathcal{L}_3$), and processing delay ($\mathcal{L}_2$).

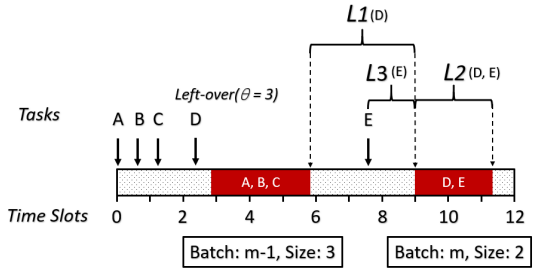


Figure 4: Delay Costs

Next, we derive $W^{(E)}_m$ of the batch B_m as follows:

$$W^{(E)}_m = \sum_{\Gamma^e_{m-1}}^{\Gamma^s_m} g(0) \cdot H_m + \sum_{\Gamma^s_m}^{\Gamma^s_m + f(|B_m|)} g(|B_m|) \cdot \mathcal{L}_2 \quad (7)$$

where $g(0)$ is the idle-time power consumption where the edge node is merely holding for the next batching without processing any tasks.

Using the loss functions $W^{(D)}_m$ and $W^{(E)}_m$, we can define the task batching problem as a constrained optimization problem:

$$\begin{aligned} &\arg\min_{B_1, B_2, \ldots, B_M} W^{(D)}_m + \lambda \cdot W^{(E)}_m, 1 \leq m \leq M \\ &s.t., |B_m| \leq \Theta \end{aligned} \quad (8)$$

where λ is a weighting factor that balances the importance of delay and energy. λ is often defined by the application to reflect its emphasis on delay minimization (lower λ value) or energy savings.

In our problem, we need to predict the batching size in real-time and cannot observe the task arrivals in the future. Therefore, variables such as the future arriving tasks A_m and task holding time H_m are unknown. To address this challenge, we design an integrated offline-online regret minimization framework to dynamically decide the optimal batch size. In particular, we start with an initial batch size. After the batch has been processed, we "look back" to see if another batch size would be better (referred to as "regret") in terms of the combined loss function defined in Equation (8) through an offline evaluation phase. Based on the regret, we adjust our batch size in the future so the regret can be minimized using an online learning phase. We elaborate on the two phases below.

Offline Evaluation Phase: the offline phase leverages the historical data to evaluate the mistakes (regret) that the current batching strategies have made. In particular, we derive the regret function $\mathcal{R}$ as:

$$\mathcal{R}_{|B_{m-1}|} = \sum_{m'=1}^{m-1} \tilde{W}^{(D)}_{m'} + \lambda \cdot \tilde{W}^{(E)}_{m'} - W^{(D)}_{m'} - \lambda \cdot W^{(E)}_{m'} \quad (9)$$

where $W^{(D)}_{m'}$ and $W^{(E)}_{m'}$ denote the delay and energy loss if *optimal* task batching strategies $\tilde{\Gamma}^s_1, \tilde{\Gamma}^s_2, ..., \tilde{\Gamma}^s_M$ were picked. The optimal strategies are derived by solving Equation (8) using a genetic algorithm [26]. $\tilde{W}^{(D)}_{m'}$ and $\tilde{W}^{(E)}_{m'}$ denote the delay and energy loss associated with the *actual* task batching strategies we had made in the past.

Online Learning Phase: after deriving the regret, we design an online learning phase to guide the next batching strategy to be as close to the optimal solution as possible. Formally, we assume a set of available actions $\mathcal{A}$ to be taken for the next batch. Here, the actions are defined as the available batch sizes i.e., $\mathcal{A} = \{1, 2, ..., \Theta\}$. We use a weight vector $\mathcal{W} = \{w_1, w_2, ..., w_\Theta\}$ to represent the probabilistic distribution for the action set, where $w_i (1 \leq i \leq \Theta)$ is the probability of choosing batch size as i as the next strategy. The weight vector follows the constraint $\sum_{w \in \mathcal{W}} = 1$.

Given the above definitions, we derive an accumulated regret as:

$$\mathcal{R} = \sum_{i=1}^{\Theta} (w_i \cdot \mathcal{R}_i) \quad (10)$$

The accumulative regret represents the extra cost compared to the cost achieved by the optimal batching size. The goal of the online learning phase is to dynamically update $\mathcal{W}$ so that the overall regret $\mathcal{R}$ can be minimized. We develop an online regret minimization scheme by extending the PROD algorithm [52]. We present our algorithm in Algorithm 1. The above regret minimization algorithm allows the regret to be bounded by $O(\sqrt{\ln \Theta \cdot T})$. We refer to the proof in [52].

Algorithm 1 Online Regret Minimization

Input: weight vector $\mathcal{W} = \{w_1, w_2, ..., w_\Theta\}$, learning parameter η, current batch index m
Output: updated weight vector $\mathcal{W}' = \{w'_1, w'_2, ..., w'_\Theta\}$
for all $t \in [0, T]$ **do**
  if t is a control point **then**
    for all $i \in [1, \Theta]$ **do**
      Normalize $p_i = \frac{\eta \cdot w_i}{\sum_{i=1}^{\Theta} (\eta \cdot w_i)}$
    end for
    Update $\mathcal{R}_{B|m-1|} = \mathcal{R}_{B|m-1|} \cdot p_i$
    for all $i \in [1, \Theta]$ **do**
      $w'_i = (w_i \cdot (1 + \eta \cdot R_i))^{\frac{\eta}{\eta - 1}}$
    end for
  end if
end for
Return $\mathcal{W}'$

The offline evaluation phase is performed periodically due to its computational complexity (period setup discussed in Section VI). We refer to each period as a *control period* and the beginning of each period as a *control point*. The online learning phase is performed after each batch processing is finished.

B. Optimal Contracting with Asymmetric Information

The SOTB module above locally optimizes the trade-off between delay and energy cost on a single node. This module alone cannot satisfy the optimization problem defined in Equation (1) because global control of the system is necessary to provide the optimized QoS for the application. Consider a scenario where a manufacturer node receives too many tasks from its suppliers. The manufacturer node can be overloaded and fail to meet QoS requirements of the application as well as encounters high energy consumption. On the other hand, a manufacturer node that receives too few tasks would fail to fully leverage the idle GPU resource on the node. We found global control mechanisms that dynamically adjust the task offloading (i.e., the supply chains) of edge nodes is crucial in addressing this issue. A key challenge of designing such a global control scheme lies in the information asymmetry where no party in the system is allowed to have full information of all edge nodes. To this end, we design a decentralized Optimal Contracting model with Asymmetric Information (OCAI) scheme that allows edge nodes to negotiate and build supply chains autonomously without revealing their private information. In particular, the OCAI consists of two interactive processes: 1) a manufacturer node first evaluates its operation status and decides whether to take more tasks from the application and at what quantity through a *resource listing process*; 2) the suppliers observe the requested tasks from the manufacturers and decide which manufacturer to offload task to through a *bidding process*.

Resource Listing Process: the resource listing process evaluates the utilization of an edge node (whether it is overwhelmed by too many tasks or it is underutilized) and decides how many more tasks the edge node will take. This is a challenging decision problem because the operational status (GPU usage, CPU usage, memory) is quite dynamic and hard to predict in the IES. For example, a smartphone owned by a user can be idle when the user is charging the phone but very busy when the user is using the phone apps. Therefore, it is difficult for an edge node to decide whether it will be capable of taking more tasks in the future.

Luckily we found the problem can be nicely mapped to an inventory model in economics [46]. In particular, the inventory model studies the problem of whether an inventory should be refilled and at what quantity if so given unpredictable demands. We map our problem as follows: the total number of tasks that an edge node processes is the "inventory size", denoted as V. The number of tasks that the edge node can finish per time slot is the "demand", denoted as D. The goal is to define a threshold R, and a refill size Q, so that every time the inventory size drops below the threshold R, a new order of Q tasks should be issued to satisfy future demands in the system.

We solve this problem by extending the classical (Q, R) reordering model [53] that can jointly derive the optimal Q and R values. Assuming the edge node needs to reorder Q tasks, the associated cost $C(Q)$ is derived as:

$$C(Q) = \lambda_h \cdot (R + \frac{Q}{2}) + \frac{\lambda_k \cdot D}{Q} + \lambda_p \cdot \frac{n(R) \cdot D}{Q} \tag{11}$$

where parameter λ_h is the holding cost per tasks, which represents the average delay cost of each task that has not been processed, including the newly added Q tasks and remaining R tasks. Parameter λ_k is the fixed cost for each order, which is the execution delay of the (Q, R) model. Intuitively, the smaller the Q is, the more likely the edge node will need to refill again in the near future, causing extra execution delay. Parameter λ_p is the cost per idle task if the inventory cannot satisfy demand (i.e., the edge node is not fully utilized). $n(R) = \int_R^\infty (x - R)\,\mathrm{d}x$ is the expected idle tasks per time slot.

Taking partial derivatives of the Equation (13) with regard to Q and R, we get:

$$\begin{aligned} \frac{\partial C(Q)}{\partial Q} &= \frac{\lambda_h}{2} - \frac{\lambda_k \cdot D}{Q^2} - \frac{\lambda_p \cdot D \cdot n(R)}{Q^2} \\ \frac{\partial C(Q)}{\partial R} &= \lambda_h - \frac{\lambda_p \cdot D \cdot (1 - F(R))}{Q} \end{aligned} \tag{12}$$

where $F(R) = \int_R^\infty x\,\mathrm{d}x$, denoting the probabilistic distribution of R. By making the partial derivatives as 0, we can derive the close-form solution as:

$$Q = \sqrt{\frac{2D \cdot (\lambda_k + \lambda_p \cdot n(R))}{\lambda_h}},\ F(R) = \frac{1 - Q \cdot \lambda_h}{\lambda_p \cdot D} \tag{13}$$

The optimal Q and R in above form can be found using the iterative algorithm in [53].

Note that the resource listing process is performed on all edge nodes that can process CNN tasks (i.e., the ones with GPUs). This design allows suppliers with idle GPU resources to serve as manufacturers as well. After deriving Q and R, we develop a new bidding algorithm that allows the suppliers to pick the best manufacturer for task offloading.

Bidding Process: the bidding process allows the supplier and manufacturer to identify the optimal offloading strategy using a game-theoretic framework. In particular, the manufacturer lists a bid of Q every time its inventory is less than R, meaning it can take Q more tasks from the suppliers. The suppliers will then compete with each other to bid for the Q tasks. Note that we assume the manufacturers will have no information about the supplier's status during bidding.

We design a game-theoretic bidding scheme that allows each supplier to selfishly pick the task that maximizes its own utility while taking into account the other suppliers' offloading strategies. We first define a We first define a delay and energy-aware *utility* function for a supplier E_i to offload task to E_j as $U_{i,j}$, which is calculated as:

$$U_{i,j} = -\frac{(\pi_{\text{energy}} + \lambda \cdot \pi_{\text{delay}}) \cdot Q_j}{\mathcal{N}_j} \tag{14}$$

where π_{energy} and π_{delay} denote the transmission energy and transmission (i.e., data offloading) delay for device E_i to send a task to E_j, respectively. Q_j denotes the number of tasks posted by E_j, and $\mathcal{N}_j$ is the *congestion rate* representing the

number of suppliers that are competing for E_j's tasks. The intuition of the above utility function is as follows. The cost term $\pi_{\text{energy}} + \lambda \cdot \pi_{\text{delay}}$ guides the suppliers to pick the nearby manufacturers that can minimize the communication delay and transmission energy. The factor $\frac{Q_j}{\mathcal{N}_j}$ further takes into account the decisions from competing suppliers and reduces the chance of picking a manufacturer whose requested tasks have already been claimed by many other suppliers.

Based on the utility function, each supplier will pick the offloading strategy that gives the highest utility until an ϵ-Nash Equilibrium [54] is reached. We say a ϵ-Nash Equilibrium is reached if any of the supplier nodes cannot further increase utility by ϵ by unilaterally changing its strategy from $U_{i,j}$ to $U'_{i,j}$, i.e., $U_{i,j} \geq U'_{i,j} + \epsilon$. The challenge for this step is that no suppliers can estimate the decision of others (due to the information asymmetric), making it difficult for them to make the best responses. Therefore, we design a decentralized Nash Equilibrium solution based on a fictitious play negotiation scheme [19]. This scheme only requires the estimation of the congestion rate $\mathcal{N}_j$ of each task. We summarize our algorithm in Algorithm 2.

Algorithm 2 Bidding Scheme

Input: A set of suppliers $S = \{S_1, S_2, ..., S_I\}$; task listings $Q = \{Q_1, Q_2, ..., Q_J\}$, decay factor $\mu \in (0, 1]$
Output: offloading strategy for each supplier $Z = \{z_1, z_2, ...z_I\}$.
Initialize: convergence flag $converge \leftarrow False$
while $converge \neq True$ **do**
  for $S_i \in S$ **do**
    S_i finds manufacturer $z_i = j'$ based on minimizing Equation (14)
    S_i sends z_i to the edge server, receives convergence flag
    Server updates $\mathcal{N}_{j'}$ +=1
    S_i receives congestion rates $\mathcal{N}_j,\ \forall 1 \leq j \leq J$
    S_i predicts $\mathcal{N}_j \leftarrow \mu \cdot \mathcal{N}_j + (1 - \mu) \cdot \mathcal{N}_j,\ \forall 1 \leq j \leq J$
  end for
end while
Return Z

We further clarify and discuss the optimality of the OCAI algorithm. First of all, we found a globally optimal solution to our problem is both intractable (proven to be NP-Hard in [27], [55], [56]) and impractical due to the lack of global device information in IES (i.e., the information asymmetry challenge discussed in the introduction). For example, a globally optimal solution would require the edge nodes to be fully cooperative and constantly share and synchronize their current status with the server. Given the complete status information of all edge devices in the system, a central controller (e.g., the edge server) will then be able to derive the globally optimal task offloading strategy of the system. However, such information-sharing requirement of the global optimal solution not only violates the assumption that an IES is composed mainly of privately owned IoT devices, but also causes excessive communication burden in the system.

Therefore, we divide the optimization problem into two sub-tasks: the resource listing process and the bidding process. For the resource listing process, the (Q, R) model we adopt identifies the *optimal* reorder threshold and order size [46] in the stochastic inventory reorder problem. We find an exact match from our problem to the stochastic inventory reorder problem. Consequently, the derived Q, R values in Equation (13) are optimal as well.

For the bidding process, the game theoretic approach targets at finding an *locally optimal* strategy for each edge node by reaching the Nash Equilibrium where each edge node cannot further maximize its own utilities. We understand that such local optimal strategy is not necessarily the optimal outcome of all edge nodes (i.e., the sum of utilities of all edge nodes). However, it is impractical to find such a global/Pareto optimal solution in a decentralized game theory problem where each edge node is not allowed to have access to other node's utility function (which would leak the device's private node status and require constant synchronization of all nodes' status) [[57].

Table I: Specifications of Edge Nodes

Type	CPU	GPU	Memory
Pi3	1.2 GHz Cortex-A53	N/A	1GB LPDDR2
TX2	2.0 GHz Cortex-A57	256-core Pascal	8GB LPDDR4
TX1	2.0 GHz Cortex-A57	256-core Maxwell	4GB LPDDR4
TK1	2.32 GHz Cortex-A15	192-core Kepler	2GB LPDDR3

V. System Implementation

This section presents our experimental platform and the implementation of the EdgeBatch framework.

A. Hardware Setup

We implement the EdgeBatch framework on a real-world collaborative edge computing testbed that consists of 12 heterogeneous devices: 1 Jetson TX2, 1 TX1, and 2 TK1 boards from Nvidia, and 8 Raspberry Pi3 Model B boards. Figure 5 shows the implemented hardware platform for EdgeBatch. These devices represent heterogeneous hardware capabilities (Table I). We chose TX2 board as the edge server due to its superior computing power. All devices are connected via a local wireless router. We leverage TCP socket programming for reliable data communication among edge nodes. We did not explicitly explore the fault tolerance (e.g., device failure, packet loss, signal loss) aspect of the communication problem and will investigate it in future work.

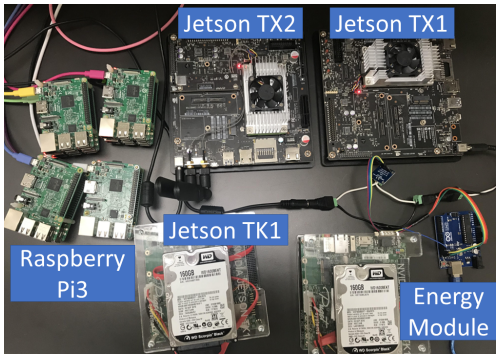


Figure 5: Heterogeneous Collaborative Edge Test Bed

B. Energy Measurement

To measure the energy consumption in our experiment, we used an INA219 Current Sensor IC, interfaced to an Arduino Uno Micro-controller board via I^2C bus to constantly monitor the current of each edge node. We multiply the current values with the default voltage (19 V for TX1 and TX2, 12 V for TK1, and 5 V for Pi3) to obtain the real-time power consumption of edge nodes.

C. System Modules and Protocol

We further describe the software implementation of EdgeBatch. All edge nodes equipped with GPU has the Tensorflow installed and execute the CNN tasks when a batch starts.

Edge Server Module: The edge server module runs on each edge server. It consists of a *bidding server program*. The bidding server program performs the bidding algorithm in the OCAI module to dynamically decide which manufacturer a supplier node should be assigned to. It coordinates the edge nodes in the negotiation phase of the bidding process (Algorithm 2) by informing the suppliers about the current congestion rate and the available tasks at each manufacturer.

Edge Modules: On each edge node (including the edge server), we develop two modules under the EdgeBatch framework: the *bidding client program*, and the *SOTB program*. The bidding client program is to find the optimal manufacturer for task offloading in the system. The client program talks to the bidding server program at the edge server to understand which tasks have been picked by other devices. If the edge node is a manufacturer, it runs the SOTB program periodically to decide the optimal batching strategy when processing incoming tasks.

VI. Evaluation

In this section, we present an extensive evaluation of EdgeBatch on the edge computing platform discussed in the previous section. We present the evaluation results through a real-world image analysis case studies: *Crowd Plate Detection*. The results show that EdgeBatch achieves significant performance gains in terms of delay and energy efficiency compared to state-of-the-art baselines.

A. Experiment Setup

Our case study *Crowd Plate Detection (CPD)* has been introduced in Section III. In particular, we use the CNN algorithm developed in [58]: a pre-trained ImageNet model VGG-16. The VGG-16 model is a state-of-the-art deep CNN with 16 layers for image recognition tasks. We use the automatic license plate recognition (ALPR) dataset [59] to emulate the collected images from IoT devices.

We choose the following baselines from recent literature. We observe that there exists no baseline that jointly addresses the task batching and task offloading issues in collaborative edge systems. Therefore, we first picked a few representative baselines for task batching schemes.

- **No Batching (NB)**: the images are processed one by one by the GPU on edge nodes.
- **Fixed Size Batching (FS)**: a heuristic batching scheme where the batching is performed whenever the number of arrived images reaches α_1.
- **Fixed Period (FP)**: a heuristic task batching scheme where the batching is performed periodically with a period of α_2.
- **Bus Waiting (BW) [60]**: a dynamic task batching scheme used in solving the optimal bus waiting time.
- **Online Learning (OL) [61]**: an online learning-based batching algorithm that dynamically adjusts the batch holding time using an online feedback control mechanism.

We also choose a few state-of-the-art baselines that manage the task offloading of the edge computing systems for a fair comparison with our scheme.

- **BGTA [27]**: A bottom-up task offloading scheme that allows manufacturers to selfishly compete for tasks to maximize utilities.
- **TDA [36]**: A top-down task offloading scheme that uses Mixed Integer Linear Programming (MILP) to minimize the deadline miss rate of tasks.
- **CoGTA [20]**: A recent task offloading scheme that allows edge nodes to trade tasks using a negotiation scheme that satisfies the delay and energy requirements of the application.

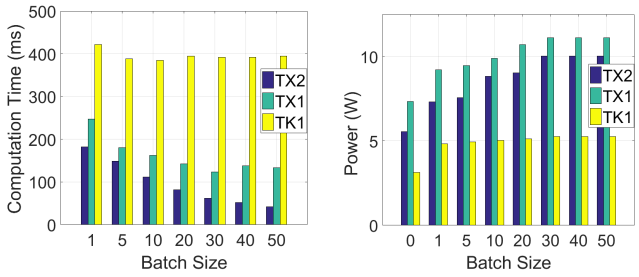
Note that both BGTA and TDA baselines do not allow IoT devices to offload tasks to each other. Instead, they let IoT devices offload tasks to edge servers (i.e., the TX2 board). We combine the task batching and offloading baselines as follows: we first run the task offloading scheme to dynamically organize the task flows of the collaborative edge system. We then run the task batching scheme to identify the optimal batch size for each edge node that has received offloaded tasks.

We have an initial bootstrapping phase to tune the parameters in the systems of the above schemes. We run the experiment for 100 time slots and find the parameters that give the minimum delay and energy costs (based on Equation (8) in Section IV). In particular, we set α_1 = 4, α_2 = 200 ms for FS and FP baselines, respectively. We set the duration for a time slot as 100 ms and the control period for the SOTB and OCAI algorithms in EdgeBatch as 5 seconds.

B. Task Batching Profiling

We first profile the computation time and energy cost functions $f(\cdot)$ and $g(\cdot)$ (defined in Section IV) with a varying batching size. We observe that, in general, increasing the batching size would reduce the average computation time of the images because of data parallelization (Figure 6(a)). The average delay slightly increases for TX1 and TK1 when the batch size exceeds 20 and 40 respectively. This is because the large batch sizes overload the GPUs on the two nodes and eventually delay the tasks. The above observations reiterate the potential benefit of leveraging large batch size to improve the delay of the tasks. We also found that devices with higher-end GPUs (i.e., TX1 and TX2) have significantly more

improvement than the device with lower-end GPU (i.e., TK1). This observation highlights the importance of considering the heterogeneity of the edge nodes in the design of an optimal task batching strategy.



(a) Batch Size vs Computation Time (b) Batch Size vs Power

Figure 6: Edge Node Profiling vs Batch Size

We also observe that increasing the batch size would increase the power consumption of an edge node (Figure 6(b)). The power consumption increase is also observed to be more significant on the devices with higher-end GPUs (i.e., TX1 and TX2). The goal of EdgeBatch is to identify the optimal batching strategy that best optimizes the above delay-energy trade-off (i.e., Equation (8) in Section IV).

C. End-to-End Delay

In the next set of experiments, we evaluate the end-to-end (E2E) delay of all the compared schemes. We run the experiment 100 times to generate the results and each run consists of 1000 time slots. Figure 7 summarizes the E2E delays of all the combinations of baselines and EdgeBatch. We show both the average delay and the one standard deviation of the results. We observe that the EdgeBatch scheme has the least E2E delay and the smallest standard deviation compared to the baselines. Compared to the best-performing baseline (CoGTA+OL), EdgeBatch achieved 31.1% decrease in E2E delay. We attribute such a performance gain to 1) the task batching module (SOTB) that fully utilizes the data parallelization of the GPUs to save average processing time of the CNN tasks; and 2) our task offloading algorithm (OCAI) that finds the optimal supply chains that allow the edge nodes to search for the most efficient way to collaboratively finish CNN tasks. We also observe that the schemes without batching have significantly longer E2E delays compared to schemes that adopt task batching. This again highlights the importance of task batching for DNN applications at the edge.

An important concern regarding the EdgeBatch scheme is the communication overhead of the data offloading tasks in the supply chains. We found in Figure 7 that the data offloading overhead for all compared schemes is relatively small compared to the overall E2E delay of the tasks. We also observe that the BGTA and TDA schemes have higher communication overheads than our scheme and CoGTA.

To further evaluate the effect of the communication tasks for data offloading, we compare EdgeBatch with all the task offloading baselines by gradually increasing the number of tasks per time slot. The results are shown in Figure 8. We

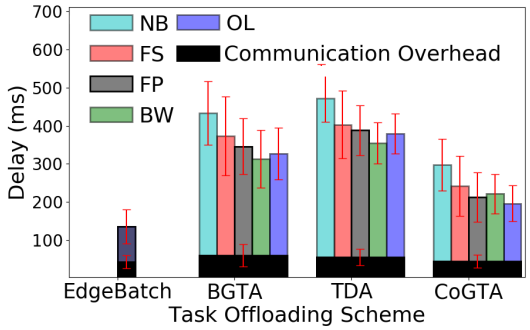


Figure 7: Average E2E Delay for All Schemes

found that the BGTA and TDA schemes consistently have significantly higher communication overheads than our scheme and CoGTA. This is due to the fact that both BGTA and TDA offload all tasks that an edge node cannot finish to the edge server, which becomes the bottleneck for the data offloading requests. In contrast, both EdgeBatch and CoGTA achieve a much lower overhead because they are able to distribute the data offloading tasks by performing peer offloading (i.e., IoT devices can offload DNN tasks to each other). While EdgeBatch has a similar data offloading overhead as CoGTA, it is superior because its optimized batching strategy allows IES system to perform DNN tasks much faster (as shown later in Figures 9 and 10).

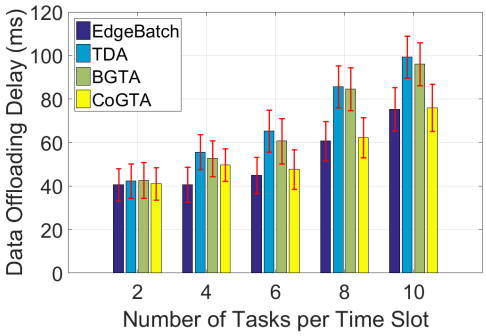


Figure 8: Average Data Offloading Overhead for All Schemes

We further evaluate the E2E delay with respect to the task frequency (i.e., number of tasks per time slot) in Figure 9. We compare EdgeBatch against the best-performing task offloading scheme (i.e., CoGTA) coupled with different task batching schemes. We observe that EdgeBatch significantly outperforms all baselines in various task frequency settings. This shows that EdgeBatch is more robust than the baselines when the workload in the system changes. We attribute this performance gain to the adaptive design of the SOTB scheme that can adjust the batching size in real-time with uncertain arrival rate of the tasks.

We also evaluate the deadline hit rate (DHR) of the application under different deadline requirements of the system. The DHR is defined as the ratio of the CNN tasks that have finished within the deadline. We fix the task frequency as 6 tasks per time slot and gradually relax the deadline requirement. The results are presented in Figure 10. We observe that EdgeBatch significantly improves the DHR compared to all schemes

and is the first one that reaches 100% DHR as the deadline increases. This result further showcases that EdgeBatch can better satisfy the real-time requirements of the application.

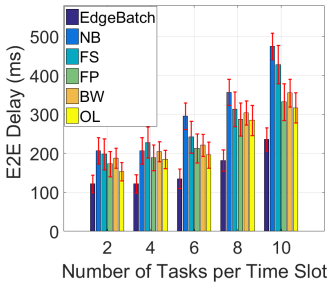


Figure 9: E2E Delay in CPD

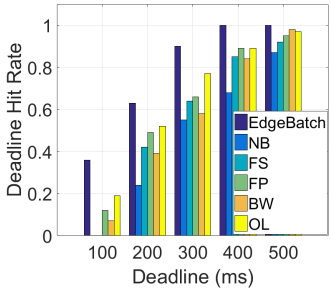


Figure 10: DHR in CPD

D. Energy Consumption

In the next set of experiments, we focus on the energy consumption of edge nodes. As mentioned in Section IV, the energy consumption is normalized to reflect the proportion of battery that is consumed by a scheme to accomplish all tasks [19]. The results of the average normalized energy consumption on edge nodes are shown in Table II. We can observe that EdgeBatch consumes significantly less energy as compared to all other baselines except TDA. TDA consumes the least amount of energy because it tends to push all the tasks from the suppliers to the edge server (i.e., the TX2 boards). In another word, the TDA scheme under-utilizes the diverse resources on the edge devices and pushes all extra computation burden to the server nodes. Such a task offloading strategy gives TDA scheme the largest amount of delay as shown in Figure 7-8.

Table II: Normalized Energy Consumption Comparison

	TX2	TX1	TK1	Pi3	Overall
EdgeBatch	0.802	0.781	0.753	0.633	8.470
BGTA + NB	0.893	0.875	0.879	0.675	9.344
BGTA + FS	0.874	0.865	0.855	0.675	9.238
BGTA + FP	0.852	0.866	0.847	0.675	9.180
BGTA + BW	0.887	0.858	0.840	0.675	9.221
BGTA + OL	0.824	0.827	0.829	0.675	9.011
TDA + NB	0.923	0.903	0.803	0.553	8.576
TDA + FS	0.885	0.864	0.755	0.553	8.326
TDA + FP	0.857	0.841	0.724	0.553	8.162
TDA + BW	0.862	0.852	0.748	0.553	8.242
TDA + OL	0.833	0.839	0.719	0.553	8.100
CoGTA + NB	0.906	0.880	0.843	0.682	9.350
CoGTA + FS	0.875	0.853	0.851	0.682	9.251
CoGTA + FP	0.837	0.832	0.819	0.682	9.068
CoGTA + BW	0.842	0.866	0.830	0.682	9.168
CoGTA + OL	0.822	0.817	0.828	0.682	9.026

"Overall" is the sum of normalized energy consumption of all edge nodes.

VII. Conclusion and Future Work

This paper presents the EdgeBatch framework to support DNN applications in Intelligent Edge Systems. We develop a novel task batching scheme for GPU-enabled IoT devices that significantly accelerates CNN-based image detection tasks at the edge. We also design a new task offloading scheme by extending the supply chain and game theory models to coordinate the collaboration among edge nodes such that the QoS of the application is optimized. We implemented the EdgeBatch framework on a real-world heterogeneous edge computing testbed and evaluate it through a real-world DNN application. The results demonstrate that EdgeBatch achieves significant performance gains compared to state-of-the-art baselines in terms of reduced delay and improved energy efficiency.

Our model has some limitations that deserve further investigation. First, the dynamic feature of IoT network may affect the distributed sensor and processing nodes setup (e.g., the availability of some edge nodes may be intermittent due to the mobility issue). EdgeBatch can be readily extended to handle this issue by leveraging two core designs in the system. First, the OCAI module of EdgeBatch is designed to generate dynamic supply chain graphs through a game-theoretic process. In this process, each edge node evaluates the dynamic communication overhead, available computing resources, and energy consumption, and self-organize into a dynamic supply chain that minimizes communication delay and energy cost. In this design, if an edge node becomes unavailable (e.g., has a poor network connection or travels to a distant area), it is unlikely to be selected as a manufacturer due to excessive communication cost. The second core design to address dynamics is the STOB module, which assumes no prior knowledge of the task arrival time and is agnostic about the network delay variations caused by mobility. We note such stochastic design of the STOB batching model is also robust against the dynamics of mobile devices. One particular issue we did not consider in our model is the task reassignment where a mobile device becomes unavailable and has tasks unfinished. This can be addressed by extending the OCAI module in our system to further incorporate task-reassignment and adaptive workload management strategies (such as FemtoCloud [18], DPA [62]) where unfinished tasks could be immediately migrated to other backup devices. We leave these extensions for future work.

Second, we assume a simplified task model in the EdgeBatch system where the tasks are homogeneous and pipelined. While such a simplification is common in resource management of many edge computing systems [10], in real-world scenarios, a task can be composed of multiple subtasks with heterogeneous inputs (e.g., images, texts, videos, sensor readings, etc.) where the subtasks may also have complex dependencies [20]. A potential solution to handle heterogeneous tasks is to leverage heterogeneous supply chain models where the manufactures with diversified facilities process different types of materials from the suppliers [63]. We plan to further improve EdgeBatch by considering the runtime scheduling of heterogeneous DNN tasks using frameworks such as SOSPCS [31], and incorporating more complex task modeling techniques such as MakeFlow [64] and DCSS [65] that regulate the system workflow to impose the relevant task dependencies.

Finally, this paper focuses on a particular type of deep learning task, namely CNN based image detection. While this type of task enables many killer applications in edge computing (e.g., disaster response [4], abnormal event detection [66], and traffic alert systems [27]), we plan to further test EdgeBatch

on a more diversified set of DNN techniques and application scenarios. For example, the recurrent neural networks (e.g., RNN, LSTM, GRU) and neural encoding techniques (e.g., autoencoders and network embedding) are also commonly used in IES applications (e.g., voice recognition and urban sensing [67]). We expect EdgeBatch to be able to accelerate these DNN algorithms as well since the task batching scheme in EdgeBatch explores the fundamental trade-off between delay and energy, which is common to DNN algorithms that run on GPU-enabled IoT devices [12].

REFERENCES

[1] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," pp. 31–36, 2018.

[2] D. Y. Zhang, N. Vance, and D. Wang, "When social sensing meets edge computing: Vision and challenges," 2019, accepted.

[3] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[4] X. Li, H. Zhang, D. Caragea, and M. Imran, "Localizing and quantifying damage in social media images," *arXiv preprint arXiv:1806.07378*, 2018.

[5] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," pp. 20–25, 2016.

[6] S. Bateni and C. Liu, "Apnet: Approximation-aware real-time neural network," pp. 67–79, 2018.

[7] H. Li, K. Ota, and M. Dong, "Learning iot in edge: deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[8] G. Zhong, A. Dubey, C. Tan, and T. Mitra, "Synergy: An hw/sw framework for high throughput cnns on embedded heterogeneous soc," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 2, p. 13, 2019.

[9] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[11] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang, "Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks," *IEEE access*, vol. 4, pp. 5896–5907, 2016.

[12] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework," p. 4, 2017.

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

[14] "Aws deeplens," https://aws.amazon.com/deeplens/, accessed: 2019-04-23.

[15] "Nvidia egx edge computing platform," https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/, accessed: 2019-04-23.

[16] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," p. 23, 2016.

[17] F. Farshchi, Q. Huang, and H. Yun, "Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim," *arXiv preprint arXiv:1903.06495*, 2019.

[18] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," pp. 9–16, 2015.

[19] D. Y. Zhang, T. Rashid, X. Li, N. Vance, and D. Wang, "Heteroedge: taming the heterogeneity of edge computing system in social sensing," pp. 37–48, 2019.

[20] D. Zhang, Y. Ma, C. Zheng, Y. Zhang, X. S. Hu, and D. Wang, "Cooperative-competitive task allocation in edge computing for delay-sensitive social sensing," 2018.

[21] Y. Huang, X. Song, F. Ye, Y. Yang, and X. Li, "Fair caching algorithms for peer data sharing in pervasive edge computing environments," pp. 605–614, 2017.

[22] D. Franklin, "Nvidia jetson tx2 delivers twice the intelligence to the edge," *NVIDIA Accelerated Computing— Parallel Forall*, 2017.

[23] G. Neubig, Y. Goldberg, and C. Dyer, "On-the-fly operation batching in dynamic computation graphs," pp. 3971–3981, 2017.

[24] D. Wang, T. Abdelzaher, B. Priyantha, J. Liu, and F. Zhao, "Energy-optimal batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mplatform case study," *Real-Time Systems*, vol. 48, no. 2, pp. 135–165, 2012.

[25] Y. Wang, B. Li, R. Luo, Y. Chen, N. Xu, and H. Yang, "Energy efficient neural networks for big data analytics," p. 345, 2014.

[26] Y. Xuan, J. Argote, and C. F. Daganzo, "Dynamic bus holding strategies for schedule reliability: Optimal linear control and performance analysis," *Transportation Research Part B: Methodological*, vol. 45, no. 10, pp. 1831–1845, 2011.

[27] D. Zhang, Y. Ma, Y. Zhang, S. Lin, X. S. Hu, and D. Wang, "A real-time and non-cooperative task allocation framework for social sensing applications in edge computing systems," pp. 316–326, 2018.

[28] C. L. Zitnick and P. Dollár, "Edge boxes: Locating object proposals from edges," pp. 391–405, 2014.

[29] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," pp. 1–12, 2017.

[30] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big. little multi-core processors," *arXiv preprint arXiv:1903.05898*, 2019.

[31] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1416–1427, 2019.

[32] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "From model to fpga: Software-hardware co-design for efficient neural network acceleration," pp. 1–27, 2016.

[33] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[34] A. Toma and J.-J. Chen, "Computation offloading for real-time systems," pp. 1650–1651, 2013.

[35] D. Y. Zhang, C. Zheng, D. Wang, D. Thain, X. Mu, G. Madey, and C. Huang, "Towards scalable and dynamic social sensing using a distributed computing framework," pp. 966–976, 2017.

[36] Z. Ning, P. Dong, X. Kong, and F. Xia, "A cooperative partial computation offloading scheme for mobile edge computing enabled internet of things," *IEEE Internet of Things Journal*, 2018.

[37] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," pp. 1–9, 2015.

[38] L. F. Bertuccelli, H.-L. Choi, P. Cho, and J. P. How, "Real-time multi-uav task assignment in dynamic and uncertain environments," *American Institute of Aeronautics and Astronautics*, 2009.

[39] V. Kumar, *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[40] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," pp. 63–74, 2010.

[41] S. Jagannathan, J. Nayak, K. Almeroth, and M. Hofmann, "On pricing algorithms for batched content delivery systems," *Electronic Commerce Research and Applications*, vol. 1, no. 3-4, pp. 264–280, 2002.

[42] C. Jayasundara and V. Gopalakrishnan, "Facilitating multicast in vod systems by content pre-placement and multistage batching," pp. 1–10, 2013.

[43] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched matrix computations on hardware accelerators based on gpus," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 193–208, 2015.

[44] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched cholesky factorization on a gpu," pp. 432–440, 2014.

[45] S. Du, M. Ibrahim, M. Shehata, and W. Badawy, "Automatic license plate recognition (alpr): A state-of-the-art review," *IEEE Transactions on circuits and systems for video technology*, vol. 23, no. 2, pp. 311–325, 2012.

[46] C. J. Corbett, "Stochastic inventory systems in a supply chain with asymmetric information: Cycle stocks, safety stocks, and consignment stock," *Operations research*, vol. 49, no. 4, pp. 487–500, 2001.

[47] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, 2012.

[48] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," pp. 779–788, 2016.

[49] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Transactions on Wireless Communications*, vol. 16, no. 3, pp. 1397–1411, 2017.

[50] A. Bellanger, A. Janiak, M. Y. Kovalyov, and A. Oulamara, "Scheduling an unbounded batching machine with job processing time compatibilities," *Discrete Applied Mathematics*, vol. 160, no. 1-2, pp. 15–23, 2012.

[51] A. Oulamara, G. Finke, and A. K. Kuiteing, "Flowshop scheduling problem with a batching machine and task compatibilities," *Computers & Operations Research*, vol. 36, no. 2, pp. 391–401, 2009.

[52] P. Gaillard, G. Stoltz, and T. Van Erven, "A second-order bound with excess losses," pp. 176–196, 2014.

[53] D. A. Schrady, "A deterministic inventory model for reparable items," *Naval Research Logistics Quarterly*, vol. 14, no. 3, pp. 391–398, 1967.

[54] I. Milchtaich, "Congestion games with player-specific payoff functions," *Games and economic behavior*, vol. 13, no. 1, pp. 111–124, 1996.

[55] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," pp. 927–930, 2009.

[56] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: an np-hard problem made easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, 1992.

[57] V. I. Zhukovskiy and K. N. Kudryavtsev, "Pareto-optimal nash equilibrium: Sufficient conditions and existence in mixed strategies," *Automation and Remote Control*, vol. 77, no. 8, pp. 1500–1510, 2016.

[58] H. Li, P. Wang, and C. Shen, "Toward end-to-end car license plate detection and recognition with deep neural networks," *IEEE Transactions on Intelligent Transportation Systems*, no. 99, pp. 1–11, 2018.

[59] "Number plate datasets," https://platerecognizer.com/number-plate-datasets/, accessed: 2019-04-23.

[60] L. Fu and X. Yang, "Design and implementation of bus–holding control strategies with real-time information," *Transportation Research Record*, vol. 1791, no. 1, pp. 6–12, 2002.

[61] S. Sorin, "Exponential weight algorithm in continuous time," *Mathematical Programming*, vol. 116, no. 1-2, pp. 513–528, 2009.

[62] N. Vance, M. T. Rashid, D. Zhang, and D. Wang, "Towards reliability in online high-churn edge computing: A deviceless pipelining approach," pp. 301–308, 2019.

[63] M.-L. Tseng, "Green supply chain management with linguistic preferences and incomplete information," *Applied Soft Computing*, vol. 11, no. 8, pp. 4894–4903, 2011.

[64] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," p. 1, 2012.

[65] Y. Xue, J. Li, S. Nazarian, and P. Bogdan, "Fundamental challenges toward making the iot a reachable reality: A model-centric investigation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 3, p. 53, 2017.

[66] C. Lu, J. Shi, and J. Jia, "Abnormal event detection at 150 fps in matlab," pp. 2720–2727, 2013.

[67] W. Zhou, Z. Shao, C. Diao, and Q. Cheng, "High-resolution remote-sensing imagery retrieval using sparse features by auto-encoder," *Remote sensing letters*, vol. 6, no. 10, pp. 775–783, 2015.