Enhancing Performance of Deep Learning Training and Inference on Resource-Constrained Edge Devices

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Tanmoy Sen

December 2024

Abstract

As edge computing gains prominence for its local computation advantages, deep learning (DL) training/update alongside inference on edge devices becomes increasingly relevant. Existing training methods for DL models either rely on centralized scheduling or involve the remote cloud. However, scaling DL models and managing large datasets pose challenges in the edge scenario due to the resource constraints of the edge devices. At the same time, the inference of the DL models on the edge devices can also be challenging because of the rising advantages of *on-device* processing by incorporating accelerators such as GPU, NPU, and DSPs. However, as energy-consuming, users may limit their floating point precision. Also, many edge device users are from areas where it is prohibitively expensive for manufacturers to include high-fidelity accelerators. So, low-cost edge devices are equipped with low floating point precision accelerators, sacrificing accuracy.

Simultaneously, in recent days LLMs have transformed natural language processing, enabling advanced tasks such as automated customer support, text generation, and real-time translation. However, deploying these models in real-world edge environments encounters performance bottlenecks, especially in handling KV-cache (KVC) during inference. This KVC bottleneck can lead to frequent preemptions, increased queuing, and high response latencies, all of which are particularly problematic in time-sensitive applications like autonomous systems and healthcare diagnostics. Consequently, achieving low latency while managing memory effectively is essential for enabling LLMs in edge settings, where constrained resources require optimal KVC handling to support real-time processing demands.

This dissertation focuses on minimizing the time of both the DL training and the inference on edge devices by addressing the above issues. In this dissertation, we propose systemic heuristic and Reinforcement Learning-based approaches to reduce the training or inference time without significant loss of accuracy. This work introduces a distributed training system, DMP, emphasizing Data and Model Parallelism. DMP optimizes the training structure by clustering edge devices by leveraging geographically close nodes for data sensing and running the model partitions to reduce the overall training time. Next, the dissertation introduces

another system, SROLE, employing Shielded RL for decentralized scheduling in the same data and model parallel training scenario to reduce the load of any single node in a cluster of edge devices. SROLE enables autonomous job scheduling at each edge node, mitigating resource overloading and action collisions in such data and model parallel training scenarios with the same goal of reducing training time. Additionally, we built a system for Fast, Accurate DNN Inference on Low-Cost Edges for the already trained models, which dynamically determines layer assignments across CPU and accelerator using heuristic and RL. Finally, we introduce Mitigating KV Cache Competition to Enhanced User Experience in LLM Inference (CacheOPT) for efficient LLM inference as a critical stepping stone for LLM deployment on edge devices. CacheOPT addresses the KVC bottleneck through confidence-guided KVC allocation based on response length predictions and dynamically adjusts padding to better utilize GPU memory. It also uses a profile-based method to make real-time decisions between recomputation and swapping, based on sequence length. We evaluate our approaches using well-known ML models for various applications to show generality and effectiveness in the real world.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Tanmoy Sen

Tanmoy Sen

This dissertation has been read and approved by the Examining Committee:

Yue Cheng, Committee Chair

Haiying Shen, Advisor

Yangfeng Ji, Committee Member

Lu Feng, Committee Member

Anand Iyer, Committee Member

Accepted for the School of Engineering and Applied Science:

Engineering Dean, Dean, School of Engineering and Applied Science

December 2024

To my mother, father, sisters, brother-in-law, maternal uncle, and wife for their continuous support, guidance, motivation, encouragement, and love.

Acknowledgements

This dissertation is the result of several years of dedicated effort, involving the invaluable guidance, support, and collaboration of many exceptional individuals. I express immense gratitude to my advisor, Haiying Shen, for her guidance, advice, support, and encouragement throughout my graduate career. Dr. Shen has trained me well in all aspects of an academic career, starting from doing research, exploring new ideas, investigating challenging problems, writing manuscripts, and mentoring students. Thanks, Dr. Shen for showing me the paths.

I am very much grateful to my Ph.D. dissertation committee members (chronologically): Dr. Anand Iyer, Dr. Lu Feng, Dr. Yangfeng Ji, and Dr. Yue Cheng, for their guidance and valuable suggestions that help me shape my doctoral research. I would like to thank all my wonderful research collaborators: Dr. Anand Iyer, Dr. Masahiro Tanka, Dr. Connor Mills, Dr. Nan Deng, Dr. Yuqing Yang, Dr. Srikant Bharadwaj, and Dr. Bruce Maggs. I have learned a lot from them. Many others also provided valuable feedback and discussions along the way; it has been a tremendous privilege working with and learning from them.

I would like to thank all the past and current members of the UVA pervasive communications lab. Especially, Ankur Sarker, and Sudipta Saha Shubha. Thank you for the laughs and happy days in the lab. I feel fortunate to work with talented and wonderful student collaborators: Fazle Rabbi Masum, Zakaria Mehrab, Matthew Normansell, Brian Yu, Ali Zafar, Mahir Ashbab, and Jeffrey Chen.

Thank you to my parents for their love, continuous support, and encouragement throughout my entire life. None of my achievements would have been possible without their infinite support and steely determination to give me the best possible education. My maternal uncle has also provided constant support in pursuing my degree. I extend my heartfelt thanks to my sister and brother-in-law, whose continuous support, insight, humor, and relentless push for me to transcend my boundaries have been instrumental in my pursuit of advanced knowledge.

A heartfelt thank you to my wife, Shrabani Ghosh, who has been by my side through every challenge and success of my graduate journey. I am deeply grateful for her unwavering support, especially during my toughest moments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent past, Deep Learning (DL) models have been extensively used for various applications in our daily lives ranging from transportation [19] and image/video processing [13] to healthcare applications such as detecting user action [105], emotions [51], abnormalities in vitals [9, 23] and infections [153]. A principal reason behind the success of such technological breakthroughs is the ubiquity of the large number and the wide variety of sensors [91, 128] in edge devices today–smartphones, tablets and smartwatches–that generate inputs to these ML models. The ubiquitous availability of such rich data has enabled ML models to achieve or even exceed human performance [34].

Due to their complexity and size, these Deep Neural Networks (DNNs) are memory and computationally expensive in training. On the contrary, an edge device usually does not have sufficient memory or computation resources to conduct the entire DNN model training job. Thus, a DNN is usually trained (or updated) in the cloud, then compressed and deployed on the edge nodes for inference. Such cloud-based training can generate significant delays when the network is intermittent (e.g., disaster, network congestion), and cannot provide data privacy protection [118] for sensitive applications (e.g., medical records) as the data needs to be transferred to the cloud. Finally, during the deployment of these trained models, modern edge devices, including flagship Android and Apple smartphones and tablets, are equipped with GPUs that can greatly accelerate ML tasks [32]. Modern accelerators often employ higher precision arithmetic, which leads to increased energy consumption. Consequently, applications like AWS SageMaker and Elastic Inference allow users to configure precision settings to optimize energy usage [152], thus transforming modern accelerators into low-cost accelerators (LCAs). In addition, a significant portion of edge device users resides in regions where these devices are prohibitively expensive. For instance, as of January 2023, approximately 70% of smartphone users in Africa, Asia, and South America use low-cost Android devices [35]. These budget-friendly

devices typically feature LCAs like NPUs, TPUs, or DSPs (e.g., Qualcomm Snapdragon with Hexagon AI engine). Despite their cost-effectiveness, these cheaper accelerators compromise on accuracy due to the use of low-precision arithmetic [20]. Lastly, Large Language Models (LLM) have revolutionized natural language processing for tasks like customer support, text generation, and translation. However, deploying them on edge devices faces significant challenges, especially with KV-cache (KVC) bottlenecks that cause frequent preemptions and high latencies, critical in time-sensitive fields like autonomous systems and healthcare. Efficient KVC handling is essential for low-latency, real-time LLM performance in resource-constrained edge environments.

Prior works [14, 118, 10] for DNN training on edges involve distributed training. However, distributed training on edges handles either data parallelism or model parallelism while involving the cloud at a certain stage. Data parallelism methods [118, 10] deploy replicas of an entire neural network on the edges, and these edges have their subsets of training data. Each edge processes its training data subset and synchronizes model parameters in a parameter server running on an initiator edge or the cloud. Model parallelism methods [14] divide a neural network, distribute the shallow (earlier) layers to edges and the deep layers to the cloud, and let edges communicate with the cloud for model parameters transfer from the previous layer transfer to the next layer. Data parallelism methods may sometimes not be feasible, as deploying a large DNN model on a single edge may not be feasible. In contrast, the model parallelism methods suffer from long communication delays and intermittent networks between edges and the cloud. So, it is very difficult to develop one solution that fits all for the distributed edge scenario. At the inference spectrum of these DNN models, there is a group of methods that select [116, 93, 36, 121, 37, 124, 125] from a group of pre-selected models or generate [148] new model from unit blocks based on current available resources of the device in order to meet requirements such as accuracy, deadline and energy. Another group of methods [70, 141] chooses model compression techniques prior to executing a DNN model on edge devices to find an optimal balance between latency and energy cost for specific resource constraints. A few works [52, 55, 126] tries to apply layer-wise quantization or set the parameter values in order to optimize the inference time and accuracy. Tan et al. [107], where the authors propose a Machine Learning Model Partition algorithm (MLMP) that finds the layer assignments (i.e., where to run which layer) for the execution of the model across the CPU and the GPU in order to minimize the inference time while satisfying the accuracy requirement or maximize the accuracy when satisfying the inference time requirement. However, individual input can affect the optimal schedule for inference, so such input-agnostic methods are non-optimal. Moreover, this scheme suffers from extended decision-making overhead. Recent papers on prediction-based KV-cache (KVC) allocation [146, 49] focus on optimizing cache usage by dynamically adjusting KVC based on response length predictions, aiming

to minimize memory overhead and latency. These approaches are particularly relevant for edge devices, where limited memory and processing power make efficient KVC allocation critical to achieving real-time performance and supporting LLM inference in resource-constrained environments. The response length predicted by the ML model can be inaccurate as it is difficult to know the response length beforehand because of the iteration-wise token generation of the LLMs. Consequently, such prediction-based KVC allocation suffers from either underprovision or overprovision of the KVC. It leads to reserved memory wastage since allocated KVC is not used instantly. To summarize, we can say the state-of-the-art works cannot achieve the best-possible efficiency of training or inference time and accuracy solely depending on the edge devices. This dissertation tries to overcome the lacking in these works and achieve the optimal training or inference time with certain accuracy guarantee.

1.1 Challenges and Brief Solutions

This dissertation presents works that identify the challenges of efficient training and inference on edge devices. The following chapters consider the significant challenges that arise in ensuring the high performance of training and inference on edge devices and handling them. From now on, we will interchangeably use the term edge or edge node to refer to the edge device.

1. **Resource Constraints of the Edge Devices.** Existing distributed training utilizes either model and data-parallel model training or updating approaches on the edge devices with the involvement of the cloud at a certain stage[14, 118, 10]. However, in a disaster-stricken scenario, such distributed training may not be possible when the cloud service is unavailable. If we move the whole training job to the edge devices, using either data and model parallelism is also not feasible. For data parallelism, collecting all sensed data from the entire area to one edge node and distributing dataset partitions to edge nodes is inefficient or infeasible. Model parallelism in a cluster of edge device requires frequent edge device communication, which generates high bandwidth consumption and long latency if cluster edge devices are out of the transmission range of each other.

   We innovatively notice that concurrent data/model parallelism is a better approach for edge systems than either data or model parallelism due to edge resource constraint and wide-area data sensing. Proximity-close edge devices usually sense similar environment data, and proximity-far-away edge nodes sense data from far-away areas. We can cluster proximity-close edge nodes in the network. Then, the m collected datasets in m clusters can be considered as m subsets of the whole dataset from the entire network. Then, as shown in Figure 1.1, letting each cluster train an entire DNN model replica using

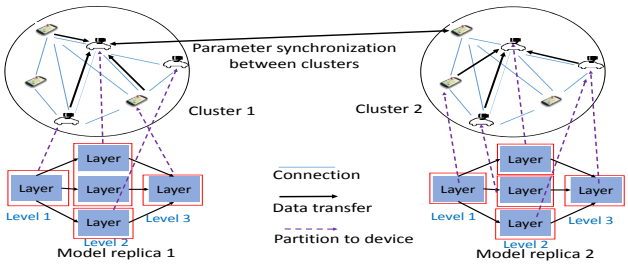


Figure 1.1: Both data and model parallelism.

the cluster's collected dataset, and partitioning the model among the proximity-close cluster edge nodes realize DMP while avoiding the aforementioned problems. Each cluster has an initiator or cluster head, which responsible for training, partitioning, and distributing the layers in the model and synchronizing model parameters to generate the final trained model. The cluster edge nodes send their sensed data to the edge device assigned with the first-layer (FLE) to collect the training data from the cluster. However, such fully distributed DNN training poses twofold challenges. First, how to divide the large DNN model and assign the partitions among the edge devices to reduce the training time? Second, how can we efficiently collect the data to train or update the training model without sacrificing accuracy?

For the first case, to find the model partition and assignment schedule that achieves the minimum training time, the dissertation introduces an optimization problem that considers the available resources of edge nodes and estimated resource demands of each layer and training data collection. Due to the high time complexity of solving the optimization problem, we propose a heuristic and a reinforcement learning (RL) based solution (Chapter 2). For the second case, if a cluster conducts training using all collected data from the cluster edge nodes, all the cluster edge nodes need to transmit their sensed data to the FLE, which consumes high bandwidth resources. We then try to avoid redundant data transmission while maintaining the accuracy of the training. We propose two methods: 1) distribution-based and 2) K-means based. n method 1, an edge node first uses Maximum Likelihood Estimation (MLE) technique [62] to efficiently get the distribution of its data sample using a subset and then sends the distribution to the FLE, and only when the FLE does not have a data sample with this distribution, the edge node sends its data sample to the FLE; otherwise, the FLE makes a replication of the data sample with the same distribution locally. In method 2, each edge node uses the k-means clustering method to compress its data sample to be sent to the FLE.

2. **Excessive Load on One Device.** In the fully distributed training approach introduced above, the cluster head assigns tasks to the edges based on the resource demands of the tasks and the available resources of the edges. Thus, the cluster head needs to continuously observe the workload conditions of all the edge nodes in its cluster. With this cluster-wide knowledge, the cluster head can avoid

overloading the edges in assigning tasks. However, such a centralized scheduling method imposes a significant workload on the cluster head, which ultimately impacts the performance of the training. Recently, RL [150] has also been used for scheduling with a similar high load. To lessen this load, this work proposes a multi-agent RL method (MARL) that enables each edge node to schedule its own jobs among its neighboring edge nodes (i.e., edge nodes in its transmission range) using RL. In MARL, each edge device works as an independent agent and makes the scheduling decision among its nearby edges, thus relieving the cluster head from the extra burden. However, without the coordination between the edge nodes, action collision may occur, in which multiple nodes may schedule tasks to the same node and overload it.

To avoid this problem, the dissertation proposes Shielded Reinforcement learning-based training on edge devices on top of the MARL-based assignment approach. The shielding approach [27] works as a separate monitor that suggests alternative actions to avoid action collision by observing the states and actions that the agents will take. In this approach, each edge node schedules its jobs using MARL, and the shield, which is deployed on the cluster head, checks the action collisions among the schedules of the edges in its cluster. Edge nodes report to the shield their action decisions, and it checks action collisions and provides alternative actions to avoid collisions.

3. **Variation of Performance among the Edge Accelerators.** Running inference of the trained models also comes with its own challenges. As we mentioned, the presence of low-precision cheap accelerators and user-defined precision settings to optimize energy usage [152] transforms modern accelerators into low-cost accelerators (LCAs). This leads to a performance difference between the CPU and the LCAs in terms of accuracy. However, the applications running on these edge devices are often time and accuracy-sensitive. A recent work called Machine Learning based Model Partition algorithm (MLMP) [107], pre-schedules a model between the CPU and accelerator offline in order to meet either the accuracy or the time deadline for such a discrepancy between the CPU and the LCAs. MLMP does such pre-scheduling without considering the input. However, the input affects the optimal schedule so such input-agnostic methods are non-optimal (Chapter 4). In addition, MLMP doesn't consider assigning the model layer-by-layer, but instead utilizes a sorting mechanism or enumerates a number of layer assignments (based on sorting) to optimize for either accuracy or latency greedily. This may lead to the skipping of better solutions.

   To address these problems, this work proposes a system that executes a model between a device's CPU and LCA for a given input. The system is based on a key observation introduced in Chapter 4 that

while LCAs give up accuracy, *individual layers of a model are affected disproportionately* regarding the accuracy loss and the execution speedup in such accelerators. Accordingly, the system places layers that are unlikely to lose accuracy but will be executed faster in the LCA and the rest of the layers in the CPU using heuristics and a RL technique.

4. **LLM KVC management.** Managing KV-cache (KVC) in large language models (LLMs) on edge devices presents significant challenges due to the limited memory and computational resources available. Edge devices often encounter high latencies from frequent KVC preemptions, particularly in real-time applications like autonomous systems and diagnostics, where rapid responses are essential. High arrival rates and limited memory capacity make it challenging to allocate KVC efficiently, leading to increased queuing and resource bottlenecks that impact overall responsiveness, making LLM deployment on edge devices difficult.

   To adresss these KVC management issues, in Chapter 5, we propose a system that leverages response length predictions to allocate memory resources more effectively. This system reduces memory overhead and enhances efficiency by dynamically adjusting the allocated KVC based on the predicted response length and confidence scores. It further improves latency with confidence-aware padding and an advanced preemption strategy, which chooses between recomputation and swapping based on system load. Together, these strategies can enable edge devices to handle high-demand LLM inference with reduced latency and can make real-time LLM applications more feasible in constrained environments in the future, as we deployed the system on GPU servers as of now.

1.2 Contributions

To summarize, this dissertation offers significant contributions addressing the challenges related to the execution of training and inference on edge devices. The key contributions of this research are as follows:

Data and Model-Parallel Training in a network of Edge Devices. In this thesis, to address the first challenge, we develop a concurrent Data and Model Parallelism deep learning (DL) system (DMP) for edge nodes in a fully distributed manner mentioned without relying on the fog or cloud, which significantly increases efficiency and scalability of DL at edge level. Chapter 2.

Efficient Load Distribution among the Edge Devices. We address the second challenge introduced in this thesis, where solely depending on one cluster overhead is insufficient. To lessen this load, we first propose MARL that enables each edge node to schedule its own jobs among its neighboring edge nodes (i.e.,

edge nodes in its transmission range) using RL. To avoid this problem, we propose Shielded ReinfOrcement learning-based DL Training on Edges (SROLE) on top of the MARL-based assignment approach in order to avoid the collisions in Chapter 3 4.

Input aware Layer Assignment between CPU and LCA during Inference. To address the challenge of performance variation between the accelerators, we evaluate low-cost accelerators for ML inference on different edge devices, and show that a static layer assignment of a model would not entail useful accuracy guarantees. Thus, we make the case for an input-specific partitioning of a model. Finally, we introduce FLEX, a lightweight and practical execution framework that dynamically finds the layer assignment between CPU and LCA for optimal latency and accuracy in Chapter4.

Mitigating KV Cache Competition to Enhanced User Experience in LLM Inference. In Chapter 5, we propose a system named CACHEOPT to address the KVC management issues in the fourth challenge by estimating the output length of a request, which guarantees that its KVC demand is satisfied with a high probability. Then, it allocates KVC to a request that equals its estimated sequence length, and reuses other requests' allocated KVC in order to avoid preemptions while reducing waiting time to satisfy the service-level-objectives (SLOs) on both Time to First Token (TTFT) and Time between Tokens (TBT). Third, instead of allocating KVC only upon request, which can lead to preemptions, it proactively allocates KVC before a request nearly exhausts its current allocation and reserves KVC globally to prevent allocation failures. It chooses a request that has long SLO for TBT, long job remaining time and short preemption time to preempt. Fifth, it selects the shortest-latency strategy between swapping and recomputation for preemptions. These components can make LLM inference more feasible on constrained edge devices in the future.

1.3 Dissertation Road Map

Chapter 2 describes our concurrent data and model parallel fully distributed training solely on the edge devices. Chapter 4 introduces our MARL scheme in the fully distributed training scenario and its further improvement using the shielding approach to avoid collisions among the individual edge devices. Chapter 3 provides details on the deadline and accuracy-aware inference on the edge devices with LCAs. Chapter 5 presents our proposed system for efficient KVC management by mitigating KV Cache competition to enhance user experience in LLM inference. Chapter 6 outlines a summary of the dissertation and discusses the potential future work.

Chapter 2

Data and Model-Parallel Training in a network of Edge Devices

The chapter considers the first challenge introduced in Chapter 1: the issue of resource constraints while executing fully distributed DL training on the edge devices. the proposed methods either run the entire model in one edge node, collect all training data into one edge node, or still involve the remote cloud. To handle the challenge, we propose a fully distributed training system that realizes both Data and Model Parallelism over a network of edge devices (called DMP). It clusters the edge nodes to build a training structure using the feature that distributes edge nodes' sense data for training. For each cluster, we propose a heuristic and an RL-based algorithm to handle the problem of partitioning a DL model and assigning the partitions to edge nodes for model parallelism to minimize the overall training time. Taking advantage of the feature that geographically close edge nodes sense similar data, we propose two schemes to avoid transferring duplicated data to the first-layer edge node as training data without compromising accuracy.

2.1 Introduction

Edge computing is widely used for diverse applications in areas such as transportation and healthcare [23]. These applications often employ machine learning (ML) frameworks using data collected by the sensors in the edge devices. ML methods have evolved into more complex and larger Deep Neural Networks (DNNs), which

This chapter is based on the publication in the Proceedings of 2023 32nd International Conference on Computer Communications and Networks (ICCCN), titled "A Data and Model Parallelism based Distributed Deep Learning System in a Network of Edge Devices" ([97]).

are memory and computationally expensive in training. An edge node usually does not have sufficient memory and computation resources to conduct the entire DNN model training job. Thus, a normal approach is cloud training, which trains (and updates) a DNN model on the distant cloud using the data collected by the edge nodes, then compresses the trained model, and finally deploys it on the edge nodes for inference. However, relying on the remote cloud may delay the update of the model. Moreover, in many cases (e.g., disaster, network congestion), the network can become intermittent [24, 73]. Consequently, sending data from the edge nodes to the cloud for training is not a suitable option [46] in such cases. Then, distributing the task of training/updating a DNN model among edge nodes becomes a promising solution. In this paper, we consider a scenario that a DNN model needs to be updated using data collected by wide-spread edge nodes in a large area. For example, sensors distributed in a geographical area cooperatively train and update a DNN model for traffic prediction using traffic data collected from the area. Althoough there are edge-friendly DL models (e.g., mobilenet, shufflenet), they sacrifice accuracy due to model compression techniques [29]. Consequently, training large DNN models on edge devices is essential when cloud datacenters become unavailable.

State-of-the-art works [14, 118, 111, 40, 21, 74, 10] for distributed training on edge nodes can be classified to three categories: data parallelism training, federated learning and cloud-edge model parallelism training. In the data-parallelism strategy [118, 40], the edge node that initializes a DNN training job has all training data and it replicates the entire neural network to its neighboring edge nodes and divides the training data among those nodes. Then, each edge node processes a training data subset and synchronizes model parameters in a parameter server running on the initiator edge node. However, it is infeasible to accumulate a large amount of data from a large area to the initiator, considering the data transmission latency and low memory and bandwidth resources. Moreover, training the entire model on a single edge node may not be feasible when the DNN model is too large to fit into one edge node. In federated learning [74, 10], each edge node downloads entire model from the cloud, updates the model using its own collected data, and synchronizes the model in the cloud. However, training the entire DNN model on a single edge node may not be feasible. Moreover, this method also suffers from the long delay of communication and intermittent network between edge nodes and cloud as explained above. Cloud-edge model parallelism training method [14, 21] partitions a neural network and distributes the shallow (earlier) layers to edge nodes, and the deep layers to the fog or cloud because the deep layers are more computation intensive [111]. The method considers the training data is already placed in the first-layer edge (FLE) and lets edge nodes communicate with fog/cloud for model parameters (weights and gradients) transfer from the preceding layer transfer to the succeeding layer. Again, it may not be efficient or even feasible to collect all data to only one edge node. Moreover, this method also suffers from the long delay of communication and intermittent network between edge nodes and cloud.

To solve the aforementioned problems, we propose a concurrent Data and Model Parallelism deep learning (DL) system (DMP) for edge nodes in a fully distributed manner mentioned in Chapter 1 without relying on the fog or cloud, which significantly increases efficiency and scalability of DL at edge level.

Model partition and assignment. A layer is a DNN unit such as a convolutional or fully-connected layer. As shown in Figure 1.1, in each model level, a model partition consists of one or multiple disjoint layers (marked by red squares), which can be executed in parallel. Edge nodes are resource-constrained in terms of computing, memory and bandwidth. Consequently, if an edge node does not have sufficient resources to conduct the computation of its assigned model partition, it becomes a straggler, which holds back the whole DNN model computation and leads to long training latency. Moreover, it is important to ensure that the FLE will not be overloaded in communication bandwidth since all cluster edge nodes need to transmit their sensed data to this edge node. To find the model partition and assignment schedule that achieves the minimum training time, we formulate an optimization problem that considers the available resources of edge nodes and estimated resource demands of each layer and training data collection. Due to high time complexity of solving the optimization problem, we propose a heuristic and a reinforcement learning (RL) based solution.

Low-overhead input data transfer. If a cluster conducts training using all collected data from the cluster edge nodes, all the cluster edge nodes need to transmit their sensed data to the FLE, which consumes high bandwidth resources. We then try to avoid redundant data transmission while maintaining the training accuracy. We propose two methods: 1) distribution-based, and 2) K-means based. We call the data sensed by an edge node in a unit time period *data sample*. In method 1, an edge node first uses Maximum Likelihood Estimation (MLE) technique [62] to efficiently get the distribution of its data sample using a subset and then sends the distribution to the FLE, and only when the FLE does not have a data sample with this distribution, the edge node sends its data sample to the FLE; otherwise, the FLE makes a replication of the data sample with the same distribution locally. In method 2, each edge node uses the k-means clustering method to compress its data sample to be sent to the FLE.

We measured the performance of the DMP system on both container-based emulation and real experiments using Raspberry-pi devices. Our evaluation shows the DMP system improves training time by 44% compared to the existing approaches while maintaining the model accuracy. We also distributed our source code [104]. The rest of the paper is organized as follows. Section 4.6 presents the related work. Section 3.4 describes our DMP system. Section 4.5 presents the performance evaluation. Finally, Section 4.8 concludes the paper with remarks on the future work.

2.2 Related Work

State-of-the-art works for distributed training on edge nodes can be classified to three categories: data parallelism training [118, 40], federated learning [10, 67, 41, 81] and cloud-edge model parallelism training [14, 21, 108]. Wang *et al.* [118] analyzed convergence rate of replicas for ML models such as Support Vector Machine (SVM) and K-means, and accordingly proposed a control method to dynamically adjust frequency of global update from each replica to the initiator in real time for minimizing the learning loss under a fixed computation resource budget for the edge nodes. Hardy *et al.* [40] devised a compression algorithm to minimize network traffic generated from transferring model parameters between local edge workers and the initiator edge to reduce communication time.

McMahan *et al.* proposed *FederatedAveraging* algorithm where the cloud combines the gradient values from the edge nodes by performing model averaging for the DNN and thus, manages to keep all the training data on each edge node to train a model. Bonawitz *et al.* [10] built a scalable production system for federated learning, using TensorFlow to ensure data privacy, and proposed new targeted network protocol to mitigate the potential synchronization overhead. Cheng and Li [41, 81] extended the approach of *FederatedAveraging* for other ML models such as decision tree and regression models.

The cloud-edge model parallelism training methods distribute one DNN across different edge nodes and cloud for training; with earlier layers being assigned to edge nodes and deep layers being assigned to the cloud [14, 108]. They predicted the gradient values related to the ML backward pass for each layer and used the predicted and selectively sent gradient value to decouple the backward pass among the DNN layers to ensure parallelism and reduce communication cost during backward pass.

2.3 System Design of DMP

2.3.1 DMP Training Structure

We assume that the edge network geographical area and the distribution of edge nodes in the area are pre-known, and the edge nodes in each cluster are within the transmission range of each other during training. To build the structure, at the initial stage, as shown in Figure 1.1, we split the entire geographical area to sub-areas so edge nodes in one sub-area forms a cluster. That is, we group the edge nodes in the entire edge node network area to multiple clusters, and each cluster is formed by proximity-close edge nodes. The cluster size (i.e., the number of edge nodes in a cluster and hence the number of clusters) is determined by the desired model parallelism degree and data parallelism degree. Higher data parallelism degree means more clusters and smaller cluster size and vice versa, and higher model parallelism degree means larger cluster size

hence fewer clusters and vice versa. The data parallelism degree depends on amount of sensed data from one cluster, and the model parallelism degree depends on the DNN size.

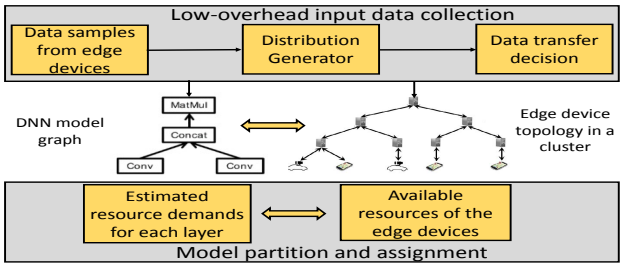


Figure 2.1: Two algorithms executed in a cluster.

In creating clusters, we ensure all edge nodes in a cluster are in the transmission range of each other so they can communicate with each other directly.

We select the edge node with the highest available capacity as the *ML initiator* for each cluster. The initiator needs to find the model partition and assignment schedule to map model partitions to the edge nodes in the cluster and initiates DNN training. It also sends the model parameters from its cluster to the global parameter server in order to accumulate all parameters across different clusters. From all initiators of all of the clusters, we select the one with the highest available resources as the parameter server. If the global parameter server cannot store all model parameters, we can use multiple edge devices as parameter servers to handle different parameters as in [48]. Figure 2.1 illustrates procedures of the proposed algorithms running in a cluster: (1) low-overhead input data collection and (2) model partition and assignment. In each cluster, each cluster edge node periodically sends to the cluster initiator its available resources. An edge node n_i's available resource is represented by R_i=$<CPU, Mem, BV>$, where BV denotes bandwidth vector and is represented by $<n_1, BW; n_2, BW; ...>$ that shows its available bandwidth to each of other edge nodes. Based on the received information of available resources, cluster initiator executes model partition and assignment.

The FLE needs to collect the training data. Depending on the system setting, it can use own sensed data or collect sensed data from other edge nodes. In the latter case, the low-overhead input data collection algorithm is used to reduce transferred data without greatly compromising training/updating accuracy. We propose two methods: 1) distribution-based method and 2) k-means based method. Here, we could only use the first input layer to collect data. However, as more collected data within a cluster helps achieve higher accuracy, we let other edge devices collect data and transfer it to the first input layer. Note that the partitions or edge nodes in one level can run simultaneously as shown in Figure 1.1. Therefore, an edge node in a preceding level needs to send parameters to the edge nodes in the succeeding level. After the last-layer edge node completes computation, a backward pass is executed. In the backward pass, the edge node assigned with the succeeding

DNN layers sends the model parameters specifically the gradients to the edge node assigned with the preceding DNN layer, and upon receiving the gradients, an edge node updates the parameters of its assigned layers. Also, in some DNNs, the last layer sends the first layer embedded data (intermediate representation of input/output), which takes a large portion of the overall data transferred between the layers [131]. After the training is completed, edge nodes within a cluster send the model parameters to initiator, which forwards the parameters to the global parameter server. The global parameter server accumulates the model parameters from all clusters and finally creates updated DNN model.

2.3.2 Model Partition and Assignment

DNN models are memory and computation extensive to fit in edge nodes with limited resources. Therefore, it is difficult to train an entire DNN on an edge node. We formulate a non-convex optimization problem to partition the entire DNN model replica in a cluster and find the optimal assignment schedule among the edge nodes in that cluster. Below, we introduce how we use prior profiling for the resource demand prediction and then present the optimization problem with heuristic and RL based solutions.

Offline Layer Resource Demand Prediction

Previous method [133] estimates the computation time of each layer, assuming the training device has sufficient computation resources. In practice, the device may not have sufficient computation resources. Therefore, we profile the resource demand and computation time for each layer and then will estimate the computation time based on the actual available resources of a device. We train regression models to predict memory and CPU demands using prior profiling of memory and CPU usage of different types of layers of a DNN. We use the *random forest regressor* and *support vector regressor (SVR)* for estimating the maximum demand of memory and CPU in the forward and backward passes, respectively. The training job of a DNN model is killed if it does not acquire the demanded memory size. Both of these regression models take the DNN layer structural parameters as input. We varied the structural parameters of a particular DNN layer structure within reasonable ranges as indicated in [133] and profile the CPU and memory usage in the forward and the backward pass. We use the TensorFlow benchmark tool [112] to profile the usage of all DL components on an edge node. We also profile the computation time of each type of layer in the forward and the backward pass separately. We use the *SVR* to estimate both the computation times for the forward and backward passes of a layer in a DNN model.

Model Graph Partition and Assignment As shown in Figure 1.1, for each DNN model replica in each geographical cluster, based on the available resources of the edge nodes in the cluster, the system divides

the entire DNN model graph into multiple partitions and then places each partition to an edge node in the cluster. Note that both the number of partitions and the partition-edge matching are dynamically determined based on the available resources of the cluster edge nodes. We formulate an optimization problem that takes both computation and communication latency in consideration and find the optimal partition and assignment schedule that generates the minimum training time. The optimization problem also considers the additional time for the data transfer from each edge node to the FLE in a cluster.

We use $\mathbf{a} = \{a_t, a_{t+1}, ...\}$ to represent the partition and assignment schedule, where a_t denotes the schedule at time slot t for layers that must start running at t. We use l_i^j to denote the i^{th} layer in level G_j and use n to denote the number of levels in the DNN model. a_t is defined as follows for each pair of layer l_i^j and edge device d_k, where $j = 1, 2..., n$ and $k = 1, 2..., |D|$, and D denotes the set of all edge nodes in the cluster:

$$a_t(l_i^j, d_k) = \begin{cases} 1, & \text{if layer } l_i^j \text{ is placed in edge node } d_k \text{ at } t \\ 0, & \text{otherwise} \end{cases}$$

We use $a_t(l_i^j)$ to denote the assigned edge node of layer l_i^j, i.e., if $a_t(l_i^j, d_k) = 1$, $a_t(l_i^j) = d_k$. After a_t is determined, the available resources of edge nodes are updated with the available resources at $t+1$, and then a_{t+1} is determined, and so on until the last layer is assigned. For simplicity, we omit t in the rest of the section.

For a layer l_i^j, we use the regression model to predict its memory demand in the forward pass and backward pass, and choose the maximum one as memory demand denoted by $T_m^{l_i^j}$. A layer l_i^j must be assigned to an edge node $a(l_i^j)$ with $M^{a(l_i^j)} \geq T_m^{l_i^j}$, where $M^{a(l_i^j)}$ denotes the available memory of the edge node $a(l_i^j)$.

We use $T^f(G_j)$ to denote the slowest execution time of a layer among the layers in level G_j or the completion time (for computation and communication) of level G_j in the forward pass. Each pair of layers between two neighboring levels (l_i^j, l_k^{j+1}) may have data transfer. The execution time of a layer in G_j includes the time for its computation and its longest data transfer time to a layer in the next level G_{j+1}. Thus, $T^f(G_j)$ is calculated by:

$$T^f(G_j) = \begin{cases} T_{dc}(l_1^1) + T_{cp}^f(l_1^j) + T_{cm}^f(l_1^j), & j = 1, \\ \max\{T_{cp}^f(l_i^j) + T_{cm}^f(l_i^j)\}, \forall l_i^j \in G_j, & 1 < j < n, \\ \max\{T_{cp}^f(l_i^j)\}, & j = n, \end{cases}$$

where $T_{dc}(l_1)$ denotes data collection time incurred to transfer the collected data from the edge nodes to the FLE within a cluster, $T_{cp}^f(l_i^j)$ denotes computation time of layer l_i^j and $T_{cm}^f(l_i^j)$ denotes maximum communication time between layer l_i^j and a layer in the succeeding level G_{j+1} in forward pass.

$$T_{dc}(l_1^1) = \max \frac{\tilde{S}_{dc}(d_k)}{B(d_k, a(l_1^1))}, \forall d_k \in D \setminus \{a(l_1^1)\}, \tag{2.1}$$

where $\tilde{S}_{dc}(d_k)$ denotes the estimated size of the data transferred from edge node d_k to the FLE $a(l_1^1)$ based on historical data and $B(d_k, a(l_1^1))$ denotes the available bandwidth between the two edge nodes at the transfer time.

To calculate $T_{cp}^f(l_i^j)$, we first predict l_i^j's computation time ($\tilde{T}_f(l_i^j)$) and its CPU demand ($\tilde{C}_f(l_i^j)$) in the forward pass, and then compute:

$$T_{cp}^f(l_i^j) = \frac{C(a(l_i^j))}{\tilde{C}_f(l_i^j)} \times \tilde{T}_f(l_i^j), \tag{2.2}$$

where $C(a(l_i^j))$ represents the available CPU resource of the edge node which is assigned with layer $a(l_i^j)$ at the running time in the forward pass.

$$T_{cm}^f(l_i^j) = \max\{\frac{\tilde{S}_f(l_i^j, l_k^{j+1})}{B(a(l_i^j), a(l_k^{j+1}))}\}, \quad \forall l_k^{j+1} \in G_{j+1}, \tag{2.3}$$

where $\tilde{S}_f(l_i^j, l_k^{j+1})$ is the predicted bandwidth demand from l_i^j to l_k^{j+1} in the forward pass using the regression model and $B(a(l_i^j), a(l_k^{j+1}))$ is the available bandwidth between the devices assigned with the two layers at the data transfer time.

We use $T^b(G_j)$ to denote the slowest execution time of a layer among the layers in level G_j or the completion time (for computation and communication) of level G_j in the backward pass. Similar to the forward pass, $T^b(G_j)$ is calculated by:

$$T^b(G_j) = \begin{cases} T_{cp}^b(l_1^j) + T_{cm}^b(l_1^j), & j = n, \\ \max\{T_{cp}^b(l_i^j) + T_{cm}^b(l_i^j)\}, \forall l_i^j \in G_j, 1 < j < n, \\ \max\{T_{cp}^b(l_i^j)\}, & j = 1 \end{cases}$$

where $T_{cp}^b(l_i^j)$ denotes the computation time of layer l_i^j and $T_{cm}^b(l_i^j)$ denotes the maximum communication time between layer l_i^j and a layer in the preceding level G_{j-1} in the backward pass.

To calculate $T_{cp}^b(l_i^j)$, we first predict l_i^j's computation time ($\tilde{T}_b(l_i^j)$) and its CPU demand ($\tilde{C}_b(l_i^j)$) in the backward pass, and then compute:

$$T_{cp}^b(l_i^j) = \frac{C(a(l_i^j))}{\tilde{C}_b(l_i^j)} \times \tilde{T}_b(l_i^j), \tag{2.4}$$

where $C(a(l_i^j))$ represents the available CPU resource of the edge node $a(l_i^j)$ at the running time in the backward pass.

$$T_{cm}^b(l_i^j) = \max\{\frac{\tilde{S}_b(l_i^j, l_k^{j-1})}{B(a(l_i^j), a(l_k^{j-1}))}\}, \quad \forall l_k^{j-1} \in G_{j-1}, \tag{2.5}$$

where $\tilde{S}_b(l_i^j, l_k^{j-1})$ is the predicted bandwidth demand from l_i^j to l_k^{j-1} in the backward pass using the regression model and $B(a(l_i^j), a(l_k^{j-1}))$ is the available bandwidth between the devices assigned with the two layers at the data transfer time.

Recall that in backward pass, for some ML models (e.g., Transformer model) the last layer needs to send the FLE the embedded data, and we use $T_{ed}^b(l_1^n)$ to denote this data transfer time.

$$T_{ed}^b(l_1^n) = \frac{\tilde{S}_{ed}(l_1^n, l_1^1)}{B(l_1^n, l_1^1)}, \tag{2.6}$$

where $\tilde{S}_{ed}$ denotes the estimated size of the embedded data based on historical data. Finally, the time latency for the backward pass is:

$$\max\{\sum_{\forall G_j \in \mathbf{G}} T^b(G_j), T^b_{ed}(l^n_1)\}, \tag{2.7}$$

where $\mathbf{G}$ denotes the set of all levels or the DNN model.

Finally, the overall goal of this optimization is to find a placement scheme that provides minimum overall training time. The objective function of the optimization is as follows.

$$O = \arg\min_{\mathbf{G}} \sum_{\forall G_j \in \mathbf{G}} T^f(G_j) + \max\{\sum_{\forall G_j \in \mathbf{G}} T^b(G_j), T^b_{ed}(l^n_1)\} \tag{2.8}$$

subject to constraint that at each time t:

$$M^d \geq \sum_{j=1}^{n}\{\sum_{i=1}^{|G_j|}\{T_m^{l_i^j} \cdot a_t(l_i^j, d)\}\}, \forall d \in D, \forall G_j \in \mathbf{G} \tag{2.9}$$

where $|G_j|$ is the number of layers in level G_j.

2.3.3 Heuristic Solution

As the formulated optimization problem is non-convex and NP-hard, we propose a heuristic to solve this optimization problem. The proposed heuristic has two phases. In the first phase, we place the first and last layers of the DNN model on one edge node [131] as the first layer and last layer share embedded data, which takes a large portion of the overall data transferred between layers. We choose the edge node with the highest available resources at time t and $t+m$ (assuming the last layer starts running at $t+m$) to place these two layers together. In the second phase, we partition and assign the rest of the DNN layers in a level-wise manner aiming to let all layers in one level complete data transfers at the same time. First, we assign the layers in each level in sequence, aiming to minimize the computation time of each level, in order to create an initial assignment schedule.

We then adjust the assignment of each layer for the optimization goal. We first build a bipartite graph $G = (L, D, E)$ where L represents the set of layers in one level, D represents the set of edge nodes within a cluster, and E represents the graph edge from the set L to D. Each edge is associated with a non-negative cost of $c(i, j)$, which is the computation time when layer i is assigned and runs on edge node j calculated by $T^f_{cp}(l^j_i) + T^b_{cp}(l^j_i)$ based on Equations (2.2) and (2.4).

We then use the Hungarian algorithm [58] to find a matching from L to D that minimizes the maximum cost among all costs. Next, we build another bipartite graph $G = (L, D, E)$ including all layers in the DNN and all edge nodes. Based on the initial assignment, we use the Hungarian algorithm to find a matching for the goal in Equ. (2.8). Basically, the algorithm reassigns each layer in every possible edge node and checks whether it reduces the cost in Equ. (2.8). If yes, it conducts the reassignment. Finally, the best possible assignment of the layers of a DNN is created.

2.3.4 RL based Partition and Assignment

The computation time overhead of the heuristic would be high with the increasing number of DNN layers and edge nodes in a cluster. To handle high overhead, we propose an RL based method. In this method, the

initiator edge node works as the RL agent to take the optimal action (level partition and assignment) upon observing the state.

State space. The state space ($\mathbf{s} \in S$) consists of the resource demands of all the layers of a DNN model, and the available resources of all the edge nodes within a cluster. For each layer, the state includes the CPU resource demand, memory demand, and its data transfer size to each layer in the next level. The state of edge nodes includes the available CPU and memory of each edge node, and available bandwidth across each pair of edge nodes at each time t. As the continuous values of these resource characteristics result in infinite size of the state space, we discretize the continuous space by dividing their value range into a number (e.g., three) of equal-width ranges.

Action space. The action space is represented by $\mathbf{a} = \{a_t, a_{t+1}, ...\} \in A$ as explained in Section 2.3.2.

Reward. Let $\mathbf{a}^t$ be the action taken (schedule is made) at that time t, then the reward function is given by:

$$\mathbf{r}(\mathbf{s}, \mathbf{a}) = \begin{cases} -\gamma, & \text{if violates Equ. (4.3)} \\ \frac{\rho}{\sqrt{O}}, & \text{otherwise (Equ. (2.8))} \end{cases}$$

where ρ is a coefficient to control the reward and γ is a large negative constant reward to ensure that a schedule violating the memory limit requirement is not valid.

We use an actor-critic RL technique [57] to find the optimal partition and assignment schedule that minimizes the training time. The actor-critic RL efficiently captures how much better an action is compared to the other actions at a given state in comparison to other RL techniques which only captures how good it is to be at a certain state [33]. Unlike [57], we used four-layer DNN for critic network as it shows good performance in our scenario. To train the RL model, we need data related to both DNN models and edge nodes. To generate the data, we use the method introduced in [133] to generate different DNN model structures and obtain their resource demands. To generate edge configuration data, we consider the number of edge nodes in the range of [2,10]. For each edge node, its CPU is chosen randomly from range [0.5,2]GHz, its memory is randomly chosen from range [64,4096]MB and the bandwidth for each pair of edge nodes is randomly chosen from range [128,1000] MBps [96]. Using the data, we trained the RL model offline for approximately 18 hours and updated using the collected data by the initiator.

2.3.5 Low-Overhead Input Data Transfer

Distribution-based Transfer Data Reduction As the edge nodes in a cluster sense similar data from the same geographical area, when an edge node's data sample (DS_2) has a similar distribution as a data

sample (DS_1) already received by the FLE, the edge node does not need to send DS_1 and the FLE makes a replication of DS_2 locally. Each edge node first estimates the type of distribution of its data sample, and then calculates the parameters of the distribution. For example, for the normal distribution, the parameters are the mean (μ) and standard deviation (σ) whereas for Poisson distribution, it is the mean. However, if the sample data is too large, an edge node may need long time to calculate the distribution parameters. To handle this problem, we use Maximum Likelihood Estimation technique (MLE) [62] that only uses a subset of the sample data to find the parameters. Since the normal distribution is the most common distribution found in the sensor data, we use this distribution as an example though it is applicable to other distributions.

MLE consists of two steps. First, the probability distribution function is converted to a strictly increasing function through logarithmic estimation:

$$\log(N(\mu,\sigma)) = -\frac{n'}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n'}(x_i-\mu)^2, \tag{2.10}$$

where n' denotes total number of data points, and x_i denotes data point i. Second, the maximization process is run on the increasing function to obtain the distribution parameters (μ and σ) following [62]. Then, each edge node calculates (μ,σ) of its sample data and sends (μ, σ) to the FLE in its cluster. As the distribution parameters represent the probability distribution, comparing these parameters of two distributions can identify whether data samples are similar.When the FLE receives (μ, σ) from an edge node, it compares it with the parameters of its received data samples in this time period. Only when none of the received data samples has similar distribution parameters, it notifies the edge node to send the data sample. Otherwise, it replicates the data sample with similar parameters. We use concept of Kullback-Leibler divergence [87], or relative entropy, to calculate the similarity between two distributions p and q, $KL(p,q)$.A lower entropy means higher similarity and vice versa. We set a threshold α and when $KL(p,q) \leq \alpha$, we consider that p and q are similar. α is set by user considering the tradeoff between training accuracy and time-efficiency for data collection.

K-means based Transfer Data Reduction

In this approach, each edge node conducts the k-means clustering on its own data sample and sends the values of cluster center and corresponding cluster size to the FLE. The cluster center is decided by minimizing the mean squared distances of each data point to its nearest center (denoted by $C(V)$):

$$C(V) = \sum_{i=1}^{m}\sum_{j=1}^{k} w_{ij}(x_i - c(j))^2, \tag{2.11}$$

where $w_{ij} = 1$ if data point x_i belongs to cluster j; otherwise, $w_{ik} = 0$, $c(j)$ represents the center for cluster j, which consists of m data points. Each edge node represents it collected data in the form of tuples $< (v_1,n_1),(v_2,n_2),\ldots,(v_m,n_m) >$, where v_i is the data value, and n_i is the number of readings with value v_i. For example, if an edge node's data sample is $\{(3,1),(4,1),(6,1),(8,1),(10,1),(12,1)\}$, using 2-means clustering, this data sample is partitioned to two clusters $\{3,4,6\}$ and $\{8,10,12\}$, with centers equal to 4.33

Table 2.1: Resource configuration for DMP experiment.

Experiment	Resource Ranges
Real edge	Mem$\in \{1024, 2048, 4096\}$MB CPU$\in \{0.25, 0.5, 1.0\}$Host Ratio BW$\in \{20, 100\}$Mbps
Container	Mem$\in \{768, 1024, 1536, 2048, 4096\}$MB CPU$\in [0.3, 1.0]$Host Ratio BW$\in \{50, 100, 200, 500, 1000\}$Mbps

and 10, respectively. Then, the compressed dataset is represented by $\{(4.33, 3), (10, 3)\}$ and is sent to the FLE.

2.4 Performance Evaluation

2.4.1 Experiment Settings

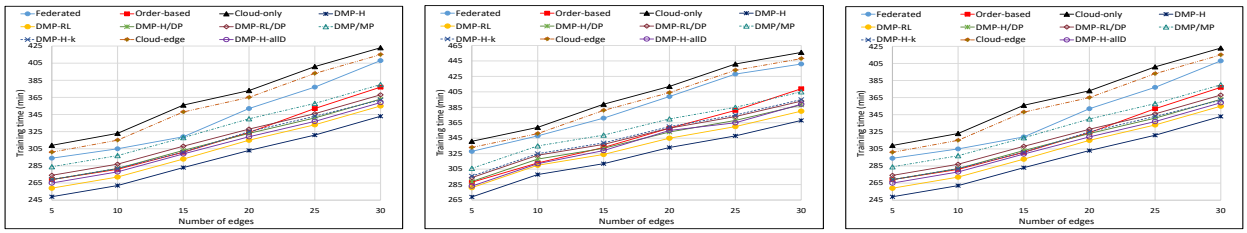
DMP uses Tensorflow for execution of the model training with parameter server strategy and is developed using keras.

Emulation. To emulate edge nodes with varying resources, we use 30 docker containers. The containers are deployed in Amazon EC2 instance of type m5ad.4xlarge.

Real experiments. Our real testbed consists of 10 Raspberry Pis; two Pis have 1 GB memory, four other Pis have 2 GB memory and four Pi has 4 GB memory. The edge nodes are connected via 2.4 GHz band wireless connection.

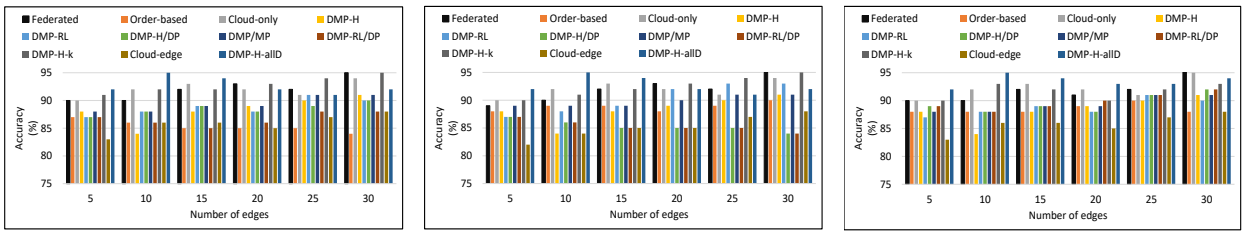
Experiment parameters. A cluster is formed by five edge nodes. The resource settings of both experiments are indicated in Table 2.1. The bandwidth connection among all the individual edge nodes to the cloud is set to the lowest bandwidth value between two individual edge nodes considering long and possibly intermittent connections. In the k-means clustering approach, we set k=3. We set ρ=1, $\gamma = 100$ for RL, and α=2 as threshold for training data collection. For all the DNN moels used in the experiment, the learning rate and batch size were set to 0.01 and 128, respectively. The mean absolute scaled error (MASE) for random forest in predicting memory demand is 4.97%. The MASE for SVRs in predicting CPU and computation times are 4.91% and 7.13%, respectively.

ML models and datasets. We run three ML models: GoogleNet Inception, VGG-16, and RNN [12]. We use the MNIST [53] dataset to run the first two models as classification tasks and the Air Quality dataset [115] for the RNN model as regression task. All these models were implemented using parameter server strategy in Tensorflow and Keras API. The baseline accuracies of the Inception and VGG models (by training on a server without any other methods) are 94% and 95%, respectively [114], and the basedline accuracy for the RNN model is



(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.2: Training time for different models in emulation.



(a) GoogleNet. (b) VGG-16. (c) LSTM.

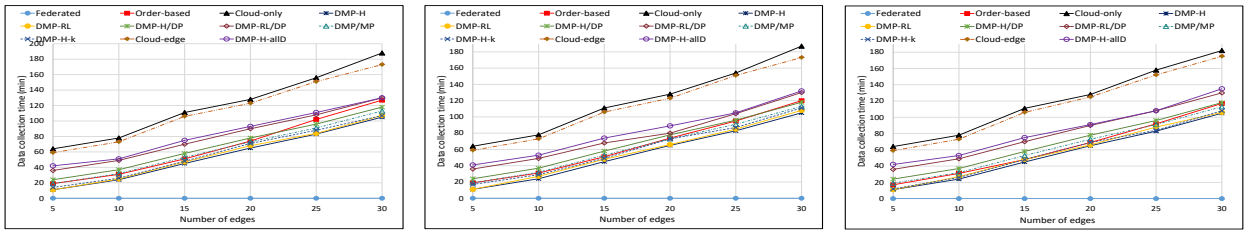
Figure 2.3: Accuracy for different models in emulation.

93%. Since there are maximally 6 clusters, We divide the dataset to 6 subsets. Each cluster has a subset as input training data and the data is randomly distributed among the edges in the cluster as their sensed data.

Metrics. Training time is the time period from the time when edge nodes send information to the FLEs to the time when the training of the whole model completes. The training data collection time is the time period from the time when edge nodes send information to the FLEs to the time when all FLEs complete preparing training data. We use 10% of the total data as the testing data on the trained model for testing accuracy, while rest of the data is used for training. We also measure the total bandwidth consumption in DNN training. The consumed bandwidth amount represents the amount of data (in Megabyte) exchanged across the whole network of devices. All of these metrics are measured for one epoch.

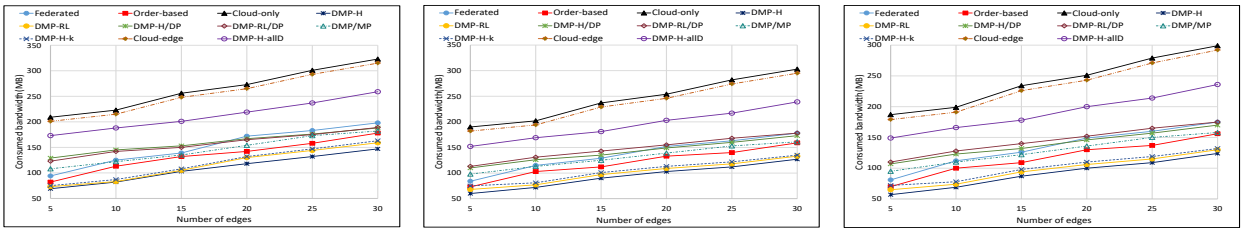
Comparison methods.

- *Federated* [10]. We first train the GoogleNet and VGG-16 model in the cloud using the EMNIST [28] and train the LSTM model using another Air Quality dataset [3]. Then, each edge node updates the trained model using its collected data, and then sends the model changes to the cloud.
- *Cloud-edge* [14]. We put the lower layers in the edge nodes randomly until each edge node's memory is full, then we put the higher layers in the cloud and ensure the last two levels are always in the cloud.
- *Cloud-only*. All the data samples collected by the edge nodes are sent to the remote cloud, where the ML model is trained.



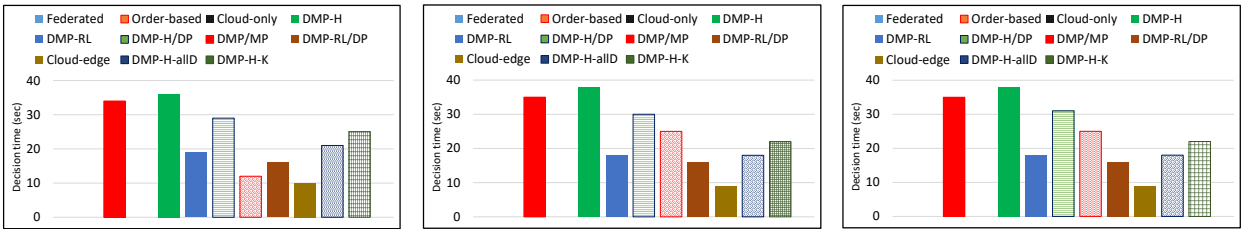
(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.4: Training data collection time for different models in emulation.



(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.5: Consumed bandwidth for different models in emulation.



(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.6: Computation time overhead of different methods in emulation.

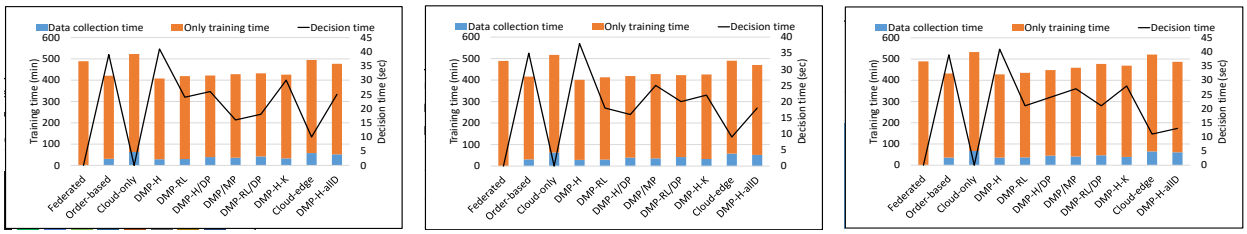
We also create several variants for DMP to show the effectiveness of different components. Unless otherwise specified, we use distribution-based transfer data reduction algorithm.

- *DMP-H*. It uses the heuristic solution.
- *DMP-RL*. It uses our RL based approach.
- *DMP/MP*. It is DMP without model parallelism, i.e., ML training is run in one randoms edge node in each cluster.
- *DMP-H/DP* and *DMP-RL/DP*. They are *DMP-H* and *DMP-RL* without the data parallelism.
- *DMP-H-k*. It is *DMP-H* using the k-means based transfer data reduction algorithm.
- *DMP-H-allD*. It is *DMP-H* in which all edge nodes in a cluster send their data samples to the FLE.
- *Order-based*. In this method, for each cluster DMP orders layers of the ML model in descending order of their memory and then CPU demand, and orders the edge nodes in ascending order of their available memory and then available CPU and performs one-to-one mapping.

2.4.2 Container-based Emulation Results

Training time. Figure 2.2 shows the training time versus the different number of edge nodes for the three models. Compared to *Federated* and *Cloud-edge*, *DMP-H* and *DMP-RL* perform 29-31%, 29-39%, and 24-36% better for the three models, respectively. Such improvement occurs due to the concurrent data and model parallelism as well as reducing the delay of data transfer. They perform 33-44% better than *Cloud-only* since *Cloud-only* needs a longer time for training data transfer to the cloud.*DMP-H* performs 4-9% better than *DMP-RL* probably because of missing some sub-optimal data points during the RL training. Both *DMP-H* and *DMP-RL* perform 8-14% better than *DMP/MP*, which shows the effectiveness of our model parallelism in reducing training time. They perform 8-15% better than *DMP-H/DP* and *DMP-RL/DP* because of the delay reduction from the data parallelism. However, *DMP/MP*, *DMP-H/DP* and *DMP-RL/DP* still outperform the comparison methods by 11-28%, which verifies the superior performance of either data parallelism or model parallelism in our system over previous methods. *DMP-H* perform 3-7% better than *DMP-H-k* since *DMP-H* avoid transferring some training data by just duplicating received data with the same distribution while *DMP-H-k* *DMP-H-k* always needs to transfer compressed data. Our *DMP-H* and *DMP-RL* show 16-28% better performance than *DMP-H-allD* for all models, which shows the effectiveness of our transfer data reduction schemes in reducing latency. They perform 17-29% better than *Order-based* for all the models. This is because our methods aim to minimize the training time while *Order-based* only assigns the layers aiming to satisfy their resource demands..

Accuracy. Figure 2.3 shows the training accuracy versus the different number of edge nodes for the three models and they achieve the baseline accuracies mentioned above. *DMP-H* and *DMP-RL* achieves similar accuracy. They increase the accuracy of *Cloud-edge* by 3%, 5%, and 8%. This is because of two reasons. First, there may be some data loss when edge nodes (widely distributed in the network) transfer training data to the single FLE. Second, there may be some data loss when training edge nodes send their parameters to the remote cloud. The results verify the advantage of conducting the model updating locally and the edge node clustering for data parallelism in our approaches. We also see that *DMP-H* and *DMP-RL* decrease the accuracy of *Federated* by 2-4%, and of *Cloud-only* by 3-5%. Without using the schemes, our *DMP-H-allD* performs similar to *Federated* and *Cloud-only* while still achieves 11-24% lower training time than them. *DMP-H* and *DMP-RL* show 1-3% lower accuracy than *DMP-H-allD* but gain 16-28% lower training time due to the transfer data reduction schemes. The system can adaptively choose which scheme should be used or adjust the threshold parameters in the schemes based on its needs in the tradeoff between accuracy and training time.



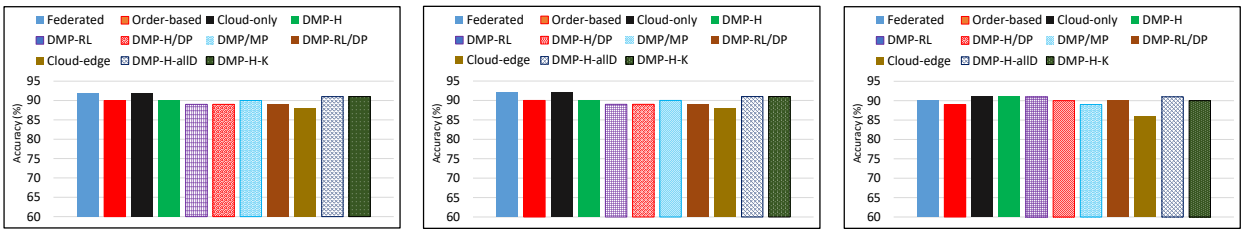
(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.7: Training time and Computation overhead for different models in real experiment.

DMP/MP performs similarly as *DMP-H* and *DMP-RL*, which means that our model parallelism does not compromise the accuracy performance. *DMP-H/DP* and *DMP-RL/DP* also perform similarly as *DMP-H* and *DMP-RL*. Without data parallelism, there would be some data loss in training data transfer to only one first-layer, as *DMP-H* and *DMP-RL* already have the transfer data reduce scheme to reduce the transferred data, they achieve similar accuracy. Both *DMP-H* and *DMP-RL* perform 3-7% better than *Order-based* for all the models. Since *Order-based* does not consider the training data collection time, the FLE may be far away from some nodes, leading to data loss in training data collection and hence lower accuracy.

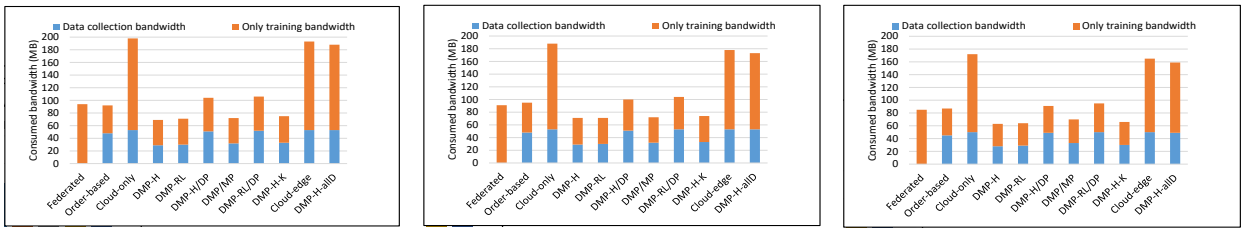
Training data collection time. Figure 2.4 shows the training data collection time versus the different number of edge nodes for the three models. *Federated* does not need training data collection since each edge node only uses its own sensed data for training. Compared to *Cloud-edge*, *DMP-H* and *DMP-RL* performs 31-32%, 24-36%, and 33-40% better for the three model, respectively. *DMP-H* and *DMP-RL* perform 31-42% better than *Cloud-only*. This is due to our transfer data reduction schemes that reduce the amount of data sent from the edge nodes to the FLE in each cluster. Also, due to data parallelism in our system, edge nodes do not have to transfer data to far-away nodes as in *Cloud-edge*. In *Cloud-only*, all edge nodes need to transfer its collected data to the remote cloud, which takes a long time. The performance of *DMP-H* is 2-5% better than *DMP-RL*. Both of them use same low-overhead data collection scheme, but the FLE chosen may be different. *DMP/MP* generates similar data collection time as *DMP-H* since they use the same data collection scheme. They both perform 14-20% better than *DMP-H/DP* and *DMP-RL/DP*. This is because without data parallelism, all edge nodes in the system need to send training data to only one FLE that may be far away from some edge nodes. *DMP-H* and *DMP-RL* perform 3-10% better than *DMP-H-k* due to the reasons mentioned above. They perform 8-11% better than the *Order-based* for all the models since *Order-based* does not consider training data collection time in layer assignment.

Consumed bandwidth. Figure 2.5 shows the consumed bandwidth versus the different number of edge nodes for all the three models. Compared to *Cloud-edge*, *DMP-H* and *DMP-RL* performs 14-22% better for all the models. *DMP-H* and *DMP-RL* perform 19-28% better than *Cloud-only*. This phenomenon occurs



(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.8: Accuracy of different methods in real experiment.



(a) GoogleNet. (b) VGG-16. (c) LSTM.

Figure 2.9: Consumed bandwidth for different models in real experiment.

due to our transfer data reduction schemes that reduce the amount of data sent from the edge nodes to the FLE in each cluster. Also, due to data parallelism in our system, edge nodes do not have to transfer data to far-away nodes as in *Cloud-edge*. In *Cloud-only*, all edge nodes need to transfer its collected data to the remote cloud, which consumes high bandwidth. The performance of *DMP-H* varies within 1% of that of *DMP-RL*. *DMP/MP* generates similar data collection time as *DMP-H* since they use the same data collection scheme. They both perform 3-7% better than *DMP-H/DP* and *DMP-RL/DP* and 2-5% better than *DMP-H-k*. This is because without data parallelism, all edge nodes in the system need to send training data to only one FLE that may be far away from some edge nodes.

Computation overhead. Figure 2.6 shows the computation time overhead on the decision making for different methods. *Cloud-edge* only involves the random decision for deciding the partition and assignment. *Cloud-only* and *Federated* do not have any decision making time as the model partition and assignment as well as data transfer are prior known. For DMP-H, we measure the time for deciding the edge nodes which need to transfer data samples to the FLE, while for DMP-H-k, the decision time includes the computation time for compressing the data sample. *DMP-RL* finds the solution 30-41% faster than that of *DMP-H*, and 28-39% faster than that of *Order-based*. *DMP-H* performs 1-3% slower than *Order-based*. *DMP-RL* has lower overhead than these two heuristics because it only needs to use the trained RL model to output the decision. *DMP-RL* performs 8-14% slower than *Cloud-edge* because *Cloud-edge* only conducts random assignment. Both *DMP-RL* and *DMP-H* performs within 10-25% limit slower than *DMP/MP* and *DMP/MP* since the latter do not have any model partition and assignment operation. However, *DMP-RL* and *DMP-H* are 1-7% slower than *DMP-H/DP* and *DMP-RL/DP*. This is because *DMP-RL* and *DMP-H* have more clusters, so the

maximum decision making time among the clusters may be longer than one fixed cluster in *DMP-H/DP* and *DMP-H/DP*. *DMP-H-k* shows 5-7% faster performance than *DMP-H*. It is because directly executing k-means clustering algorithm is faster than the process of determining whether the data needs to be transferred or duplicated in *DMP-H*.

2.4.3 Real Experiments Results

From real experiments, we observe similar performance as the emulation technique due to the same reasons mentioned above.

Training time. Figure 2.7 illustrates the training time for all the methods. Compared to *Federated*, *Cloud-edge* and *Cloud-only* *DMP-H* and *DMP-RL* perform 25-31%, 23-35%, and 33-38% better for the ML models, respectively.

Accuracy. Figure 2.8 shows the accuracy for all methods. The performance of all the methods except *Cloud-edge* are approximately similar. *Cloud-edge* performs 2-5% worse than other methods for the same reasons mentioned above.

Training data collection time. The blue part at bottom of each bar in Figure 2.7 shows the data collection time for all methods. Compared to *Cloud-edge*, *DMP-H* and *DMP-RL* perform 21-24%, 22-27%, and 21-26% better for the three models, respectively. They perform 25-34% better than *Cloud-only*.

Consumed bandwidth. The blue part at bottom of each bar in Figure 2.9 shows the consumed bandwidth during data collection time, while the orange part shows the consumed bandwidth for rest of the training period for all methods. Compared to *Cloud-edge* and *Cloud-only*, *DMP-H* and *DMP-RL* perform 14-23% better for all the three models.

Computation overhead. The black curve in Figure 2.7 shows the computation overhead on decision making time. *DMP-RL* finds solution 28-39% faster than *DMP-H* on real edge nodes.

2.5 Conclusion

Fully distributed DNN training on edge nodes utilizing both model and data parallelism is a promising way to increase the scalability of DNN model training at resource-constrained edge nodes. In this paper, we propose a DMP system to distribute layers of a large DL model onto a set of edge nodes to minimize the training time. Our emulation and real experiments show that the proposed system significantly outperforms the state-of-the-art cloud-only, cloud-edge model parallelism and federated learning approaches. In the future, we will develop

a system that chooses the optimal strategy among different choices (e.g., data/model parallelism, or both) based on real-time observations from the edge system for further improvement in training performances.

Chapter 3

Efficient Load Distribution among the Edge Devices

This chapter revisits the first challenge and tackles the second challenge highlighted in Chapter 1: excessive load on one device. It explores the challenge of having the cluster head of a cluster of edge nodes schedule all the DL training jobs from the cluster nodes. Using such a centralized scheduling method, the cluster head knows all the loads of the cluster nodes, which can avoid overloading the cluster nodes, but the head itself may become overloaded. To handle this problem, we first propose a multi-agent RL (MARL) system that enables each edge node to schedule its own jobs using RL. However, without the coordination between the nodes, action collision may occur, in which multiple nodes may schedule tasks to the same node and make it overloaded. To avoid these problems, we propose a system called Shielded ReinfOrcement learning (RL) based DL training on Edges (SROLE). In SROLE, each edge node schedules its own jobs using multi-agent RL. The shield deployed in a node checks action collisions and provides alternative actions to avoid the collisions. As the central shield node for the entire cluster may become a bottleneck, we further propose a decentralized shielding method in which different shields are responsible for different regions in the cluster, and they coordinate to avoid action collisions on the region boundaries.

3.1 Introduction

Edge devices are currently used for various applications in many areas including transportation and healthcare [9, 45, 117, 23]. These applications often deploy machine learning (ML) frameworks using data collected

This chapter is based on the publication in the Proceedings of 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), titled "Distributed Training for Deep Learning Models On An Edge Computing Network Using Shielded Reinforcement Learning" ([98]).

by the edge devices' sensors. ML models have transformed into more complex and larger Deep Neural Networks (DNNs). These DNNs are memory and computationally expensive in training due to their complexity and size. On the contrary, an edge device usually does not have sufficient memory or computation resources to conduct the entire DNN model training job. Thus, a DNN is usually trained (or updated) in the cloud, then compressed and deployed on the edge nodes for inference. Such cloud-based training can generate significant delays when the network is intermittent (e.g., disaster, network congestion), and cannot provide data privacy protection [118] for sensitive applications (e.g., medical records) as the data needs to be transferred to the cloud. In this case, distributing the job of training or updating a DNN model among only edge nodes becomes a promising solution.

Recent works [14, 118, 10] for distributed training on edges handle either data parallelism or model parallelism while involving the cloud at a certain stage. Data parallelism methods [118, 10] deploy replicas of an entire neural network on the edges, and these edges have their subsets of training data. Each edge processes its training data subset and synchronizes model parameters in a parameter server running on an initiator edge or the cloud. Model parallelism methods [14] divide a neural network and distribute the shallow (earlier) layers to edges and the deep layers to the cloud and let edges communicate with the cloud for model parameters transfer from the previous layer transfer to the next layer. The data parallelism methods sometimes may not be feasible as deploying a large DNN model on a single edge may not be feasible. In contrast, the model parallelism methods suffer from the long delay of communication and intermittent network between edges and cloud.

A concurrent data and model parallelism-based deep learning (DL) system can handle these problems. The overall goal of such concurrent data and model parallelism is to handle DL training using resource-constrained edge devices while minimizing the DL training time. In such a system 2, clusters of edges are created according to geographical locations, and each cluster trains a replica of the DL model using model parallelism based on its locally collected data. Each cluster has a cluster head that has relatively high capacity, and it assigns the partitions of a DNN model (i.e., tasks) to the cluster edge nodes with the goal of minimizing training time. Each edge within a cluster collects its own data and sends the data to the first-layer node, which collects the sensed data from all cluster edges as the training data of the model replica in the cluster. The cluster head assigns tasks to the edges based on the resource demands of the tasks and the available resources of the edges. Thus, the cluster head needs to continuously observe the workload conditions of all the edge nodes in its cluster. With this cluster-wide knowledge, the cluster head can avoid overloading the edges in assigning the tasks. However, such a centralized scheduling method imposes a significant workload on the cluster head, which ultimately impacts the performance of the training. RL [130, 150] has also been used for such scheduling in recent times with a similar high load. To lessen this load, we first propose a

multi-agent RL method (MARL) that enables each edge node to schedule its own jobs among its neighboring edge nodes (i.e., edge nodes in its transmission range) using RL. In MARL, each edge device works as an independent agent and makes the scheduling decision among its nearby edges, thus relieving the cluster head from the extra burden. However, without the coordination between the edge nodes, action collision may occur, in which multiple nodes may schedule tasks to the same node and make it overloaded.

To avoid this problem, we propose Shielded ReinfOrcement learning based DL Training on Edges (SROLE) on top of the MARL-based assignment approach. The shielding approach [27] works as a separate monitor that suggests alternative actions to avoid action collision by observing the states and actions that will be taken by the agents. In SROLE, each edge node schedules its own jobs using MARL, and the shield, which is deployed on the cluster head, checks the action collisions among the schedules of the edges in its cluster. Edge nodes report to the shield their action decisions, and it checks action collisions and provides alternative actions to avoid the collisions. However, the computational cost of a centralized shield grows dramatically with the number of edges in a cluster, and it may become a bottleneck for the entire cluster. Thus, we further propose a decentralized shielding method, in which different shields are responsible for different regions in the cluster and they coordinate to avoid action collisions on the region boundaries. Specifically, a large cluster is divided into multiple sub-clusters according to the geographical proximity, and a shield monitors the edges within each sub-cluster and communicates with its neighboring shields for the edges at the boundary of the sub-clusters to avoid action collisions, i.e., unsafe actions. The computational cost of each shield in the decentralized method is lower than that in the centralized shielding as the centralized shield's workload is distributed to a number of shields.

In summary, the contributions of this work are as follows:

- To avoid overloading a cluster head due to scheduling DL training jobs in its cluster, we initially propose a multi-agent RL-based method (MARL) that enables each edge node to use RL to schedule its own jobs. For a given DL training job, an edge node makes scheduling decisions for DNN partitions among its nearby edges depending on their resource availability to minimize training time.
- To avoid action collisions in MARL, we use the shielding approach in each cluster. The shield in a cluster collects the scheduling decisions of all edge nodes in the cluster, checks the action collisions and provides alternative actions to avoid the action collisions.
- To avoid overloading a shield in a cluster, we distribute the shielding workload to multiple shields in a cluster and each shield is responsible for a sub-cluster. The neighboring shields communicate with each other to avoid action collisions from the edges on the boundaries of sub-clusters.

- We measured the performance of the SROLE system on container based emulation on Amazon EC2 instances. Our evaluation shows that the SROLE system shows up to 59% reduction in training time and up to 48% reduction in the number of action collisions compared to the centralized RL and MARL approach. Our real device experiments also show up to 53% reduction in training time and up to 46% reduction in the number of action collisions in comparison with the centralized RL and MARL approach.

The rest of the paper is organized as follows. Section 4.6 presents the related work. Section 3.4 describes our SROLE system. Section 3.5 presents the performance evaluation. Finally, Section 4.8 concludes the paper with remarks on our future work.

3.2 Related Work

Researchers have been studying federated ML training and DNN partition distribution across cloud/fog and edge devices [14, 10, 65, 144, 66, 139, 47, 119]. In federated learning [10], edge devices update a trained model on the cloud. Individual edges download a replica of the model and update the models using their own available datasets. Finally, the updated models from individual edges are aggregated through averaging or using a control theorem on the cloud to produce the final model. Many proposed ML inference approaches partition the ML model distributed edge nodes [137, 145, 149]. The works in [137, 149] simply consider each or multiple convolution layers as a partition, and the work [145] vertically divides the convolutional layers in a convolutional neural network (CNN). The work in [14] distributes DNN partition across cloud/fog and edge devices to accelerate training or inference. The resource-constraint edge devices run lighter (earlier) layers of the model and the cloud or fog run heavier (later) layers of the model. Resilinet [134] achieves failure-resilient inference in model-parallel ML at the edge. Data parallelism training methods distribute training data among different edges for training and accumulate training updates. Wang *et al.* [118] analyzed the convergence rate of replicas for ML models such as Support Vector Machine (SVM) and K-means, and accordingly proposed a control method that dynamically adjusts the frequency of global update from each replica to the ML initiator in real time to minimize the learning loss under a fixed computation resource budget for the edges.

To efficiently run ML models on edge devices in terms of inference time, energy and memory, researchers have introduced techniques for compressing the neural networks (NNs) [102, 82, 140, 61, 6, 59, 133, 88, 72, 64, 80, 16, 94]. For example, the works in [129, 70] find compressed DNN models by formulating optimization problems that meet resource (memory, energy) constraints or minimize inference time while maximizing accuracy or reaching a specified accuracy. Han *et al.* [39] proposed cumulative pruning of the network connection, weight quantization, and compression through Huffman coding to decrease the size and inference time of an NN model without significant loss of accuracy. Ashok *et al.* [6] proposed a reinforcement learning

based policy that first removes layers from the DNN and later reduces the size of the remaining layers by deleting links. Yao *et al.* [132] used a pre-trained compressor-critic network to estimate the link weights and drop out the low-weight links.

However, there are few works on scheduling the partitions of a DNN model in model parallelism on the edge in an efficient manner. This paper addresses this problem.

3.3 Background

In the model parallelism, a layer is a DNN unit such as convolutional or fully-connected layer [79]. In each model level, a model partition consists of one or multiple disjoint layers, which can be executed in parallel. These partitions are assigned to the edge nodes based on their available resources. In this paper, we use concurrent data/model parallelism as an example to explain our proposed methods though they also can be applied to model parallelism. The problem we handle is how to schedule the different model partitions (i.e., tasks) to different edge nodes to minimize the training time. In concurrent data/model parallelism, as shown in Figure 3.1, each cluster is formed by proximity-close edge nodes. Each cluster trains an entire DNN model replica using the cluster's collected dataset, and partitions the model among the proximity-close cluster edges. The cluster head is responsible for initiating the training, partitioning and distributing the layers of the model to edge nodes, and synchronizing model parameters to generate the final trained model. In order to conduct task scheduling, the cluster head needs to continuously check the resource availability of all the edge nodes in its cluster and also estimate the resource demands of each partition in scheduling the tasks of a job. The resource availability and resource demands are for multiple resources, mainly including GPU or CPU, memory, and bandwidth.

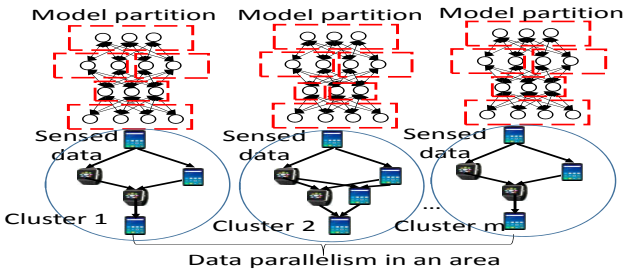


Figure 3.1: Model and data parallel ML on edges.

An edge node is considered overloaded when the sum of the resource demands of its running tasks is larger than its resource capacity for one type of resource. If an edge node running a model partition becomes overloaded, the training process may slow down. Thus, each device d_j measures its resource utilization of

each type type-k resource periodically at each timestep t as follows:

$$u_k(d_j) = \frac{D_k(d_j)}{C_k(d_j)}, \tag{3.1}$$

where $D_k(d_j)$ denotes the total resource demand of type-k resource of the tasks running on edge d_j and $C_k(d_j)$ denotes the capacity of type-k resource of edge d_j. The system pre-defines α (e.g., 0.95) and if $u_k(d_j) > \alpha$ for any type-k resource, the edge node is considered as overloaded. We define an edge node's combined resource utilization as follows:

$$u(d_j) = \prod u_k(d_j),\ k = 1, 2, ... \tag{3.2}$$

It measures the overall resource utilization across different types of resources of an edge node.

The cluster head can use RL for the task scheduling and functions as the agent in the RL. RL has three components: state, action and reward. Given the current state, the agent chooses the action that generates the maximum expected reward and receives reward for the action it takes. In this task scheduling scenario, the state is the resource demand of each layer in the DL model and the resource availability of each edge node in the cluster. The action is the schedule that assigns each partition to an edge node. The reward is defined based on the training time of the DNN model; shorter training time leads to higher reward and vice versa. Thus, using the trained RL, the cluster head observes the state and makes the scheduling decision for each DL training job. However, all of these operations create significant overhead on the cluster head. In this paper, we aim to distribute the overhead among the cluster edge nodes while decreasing training time increase due to this decentralized operation.

3.4 System Design of SROLE

3.4.1 Overview

We propose SROLE that consists of the following components.

- **Multi-agent RL (Section 3.4.2).** To distribute the job scheduling overhead on the cluster head among the cluster edge nodes, we propose a multi-agent RL-based method (MARL). In MARL, each edge node uses RL to schedule the tasks of its own DL training job without relying on the cluster head.

- **Centralized Shielding for MARL (Section 3.4.3).** Edge nodes share their neighbors since there are overlaps in the transmission ranges of neighboring edge nodes. Then, edge nodes may schedule their tasks to the same edge node since they do not know the decisions of other edge nodes, which may overload the task assigned node. Thus, we propose a centralized shielding method for MARL, which checks the action collisions and provides alternative actions to avoid the action collisions in a cluster.

- **Decentralized Shielding for MARL (Section 3.4.4).** As the centralized shield is deployed in one edge node, which may overload it, we propose a decentralized shielding method for MARL. In this method, multiple shields are deployed to the sub-clusters in one cluster and each shield is responsible for its own sub-cluster. Further, the neighboring shields communicate with each other to avoid the action collisions on the boundaries.

3.4.2 Multi-Agent RL-based Job Scheduling

The MARL method is similar to the above RL-based method, except each edge node is an agent and the state includes the resource availability of an edge node's nearby nodes rather than all the edge nodes in the cluster. The RL is initially pre-trained and distributed to each edge node. As a result, each edge node uses the RL to schedule the tasks of its own DL training jobs and keeps training the RL model. In this method, each edge takes the optimal action of assignment of the partitions based on its observed state. In particular, each edge makes its own local decision on where each layer should be assigned. Based on the decision made by each edge node, the state of the environment changes as the available resources change for the edge node where a layer is assigned. Consequently, based on the local decision taken by one edge or agent, the global decision of all agents influences the overall state. Finally, each edge node has its own long-term reward to optimize, which now becomes a function of the policies of all other agents that are updated based on the global decision. Figure 4.13 illustrates the working procedure of the multi-agent RL. Each edge node observes the state space from the environment and then takes its own action to assign partitions to itself and its neighbors, and then it receives reward. The joint action of the actions of all agents are denoted by $\mathbf{a_t^c}$: $\mathbf{a_t^c} = \mathbf{a_t^1} \bigcup \mathbf{a_t^2} ... \mathbf{a_t^i} ... \bigcup \mathbf{a_t^n}$. After the actions are taken, the state $\mathbf{s}$ and reward $\mathbf{r}$ at the next timestep $t+1$ (i.e., $\mathbf{s_{t+1}}$ and $\mathbf{r_{t+1}}$) become the state and reward at this timestep t (as indicated with arrows) for the agents to make decisions again.

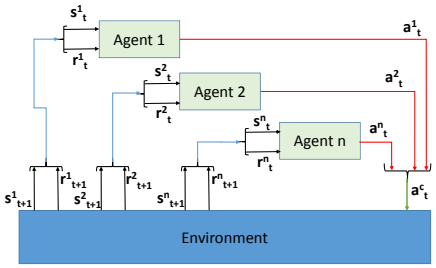


Figure 3.2: The process of multi-agent RL.

Now, we explain the corresponding state (S), action (A) and reward (R) for our proposed MARL model by each agent.

State space. The state space ($\mathbf{s_t} \in S$) consists of the resource demands of all the layers of a DNN model, and the available resources of all of a node's nearby edge nodes. For each layer, the state includes the CPU resource demand, memory demand, and its data transfer size to each layer in the next level in the DNN

model. Besides, the state also includes the utilization of the CPU, memory and bandwidth resource for each device. The state of edges includes the available CPU and memory of each edge, and available bandwidth across each pair of edges at each time t. As the continuous values of these resource characteristics result in infinite size of the state space, we discretize the continuous space by dividing their value range into a number (e.g., three) of equal-width ranges: low, medium and high.

We varied the structural parameters of a particular DNN layer structure within reasonable ranges as indicated in [133] and profile the CPU and memory usage in the forward and the backward pass. We use the TensorFlow benchmark tool [112] to profile the usage of all DL components on an edge node. The available resources on an edge device keeps changing accordingly to the layers assigned to the device.

Action space. We use $\mathbf{a_t}$ to denote $\mathbf{a_t^i}$ for simplicity. The action space represented by $\mathbf{a_t}$= $\{a_t^{i,j}\} \in A$, $(i = 1, 2, ..., |M|,\ j = 1, 2, ..., |E|)$ defines the schedule for all the layers at time t, where M denotes the set of the layers in the DNN model, E denotes the set of all nearby edges and $|\cdot|$ means the size of a set. Each element of the action space A defines which edge should be assigned with a certain layer. Action $a_t^{i,j}$ is defined as follows for each pair of layer l_i and edge device d_j.

$$a_t^{i,j} = \begin{cases} 1, & \text{if layer } l_i \text{ is placed in edge } d_j \text{ at } t \\ 0, & \text{otherwise} \end{cases}$$

After $\mathbf{a_t}$ is determined, the available resources of edges are updated with the available resources at $t+1$, and then $\mathbf{a_{t+1}}$ is determined, and so on until the last layer is assigned.

Reward. Let $\mathbf{a}_t$ be the action taken (schedule is made) at time t, then the reward function is given by:

$$\mathbf{r_t}(\mathbf{s_t}, \mathbf{a_t}) = \begin{cases} -\gamma, & \text{if memory is violated} \\ \frac{\rho}{\sqrt{O}}, & \text{otherwise} \end{cases}$$

where ρ is a coefficient to control the reward, γ is a large constant reward to ensure that a schedule violating the memory limit requirement is not valid. Furthermore, and O denotes the training time of the DNN model. After the job assignment, the states change for the next assignment and the reward is updated for all agents.

3.4.3 Centralized Shielding for MARL

The MARL method cannot ensure that none of the edge nodes get overloaded since different edge nodes may assign tasks to the same node simultaneously based on its original available resources. To handle this problem, we propose a shielding approach on top of the multi-agent RL scheme. Each cluster has a

shield deployed in the cluster head that has high resource capacity. It ensures that none of the edges get overloaded by the task assignment from all edge nodes in the cluster. In the centralized shielding method, the shield enforces the safety specification (i.e., avoiding decisions that overload an edge before being sent to the environment) during the RL learning process. After an edge node makes a scheduling decision for its job, it reports its decision to the shield in its cluster. The shield collects the decisions of all edge nodes in its cluster and checks action collisions, i.e., the actions that will make an edge node overloaded by hosting the tasks from multiple edge nodes, and then provides alternative actions to avoid the action collisions. We hope that the shield restricts MARL agents as less as possible via the minimal interference criteria. These criteria are as follows: (1) the shield only corrects joint action $\mathbf{a_t^c}$ if it violates the safety specification, and (2) the shield seeks a safe joint action $\tilde{\mathbf{a}}_\mathbf{t}^\mathbf{c}$ that changes as a few of the agents' actions as possible in $\mathbf{a_t^c}$. Figure 3.3 shows the working procedure of the centralized shielding built on top of the multi-agent RL scheme. The centralized shield observes the joint action and current state before the action is implemented. If it leads to an unsafe action, i.e., overloading of any edge, the shield suggests a safe action that will be implemented. At the same time, the shield also notifies the edges within the cluster of the safe action and assigns a constant negative reward (κ) for their originally decided action that leads to the overload of one device. Accordingly, the reward is redefined as below:

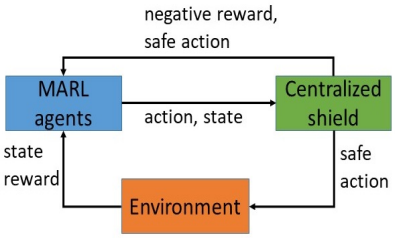


Figure 3.3: Centralized shielding.

$$\mathbf{r_t}(\mathbf{s_t}, \mathbf{a_t}) = \begin{cases} -\gamma, & \text{if memory is violated} \\ -\kappa, & \text{suggested by the shield} \\ \frac{\rho}{\sqrt{O}}, & \text{otherwise} \end{cases}$$

In the meantime, the environment changes based on the suggested safe action by the shield and the states and rewards are updated accordingly.

The shield makes a judgement about the potential violations of safety specifications. In details, the shield observes whether the joint action actually changes the resource utilization of any type of resource of an edge to a value higher than the threshold, i.e., $u_k(d_j) > \alpha$. If this condition is true (criterion (1)), there exists an action collision and the shield will choose alternative safe actions to replace the original actions in the joint action to make the edge node (say d_j) not overloaded. For each layer l_i that is assigned to edge node d_j, we

define its resource demand weight as follows:

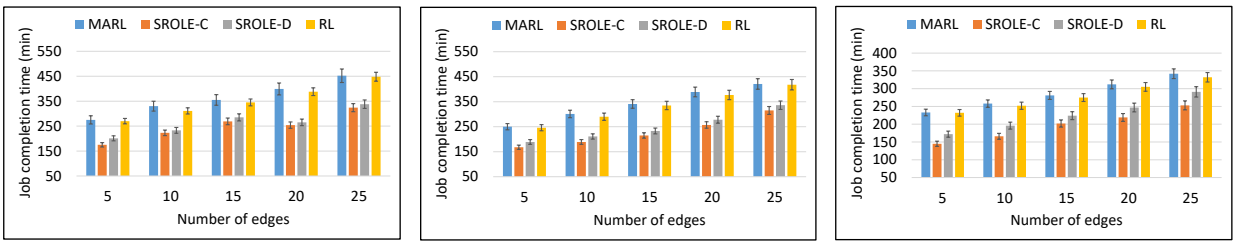
$$\omega(l_i) = \prod_k (b_k(l_i)/C_k(d_j)),\ k = 1, 2, ... \tag{3.3}$$

where $b_k(l_i)$ is the resource demand of type-k resource of layer l_i and $C_k(d_j)$ is the capacity of type-k resource of edge device d_j. While choosing the safe action, the shield first ranks the layers that are planned to be assigned to the edge node based on their resource demand weights. Then, it picks up the layer with the highest weight and finds a new host edge for it that will not be overloaded after hosting this layer. The purpose of choosing the layer of the highest weight to be rescheduled is to reduce the interference to the original joint action (criterion (2)). The shield repeats this process until the remaining layers will not overload the edge. Specifically, it searches for nearby edge nodes with high resource availability from edge node d_j, and then checks whether any of these edges can host this layer after it accepts other layers that are planned to assign to it in other actions. To quickly find such an edge node, the shield calculates the combined resource utilizations of the nearby edge nodes after they accept other planned layers assigned to them. Next, it orders the nearby edge nodes in the ascending order of their combined resource utilizations and then sequentially picks up the top node to check until it finds such an edge node. The edge node on the top generally has a high available resources and hence is more likely to be able to host the layer. After finding the new host edge, the shield creates an alternative action that assigns the layer to this edge device. As we limit our safe action from the nearby edges of the original edge node in the decided original action and ensure this newly suggested action won't overload the edge, this new action will not deviate from the previous optimal action greatly.

Algorithm 1 Pseudocode of the centralized shielding executed by the shield in a cluster.

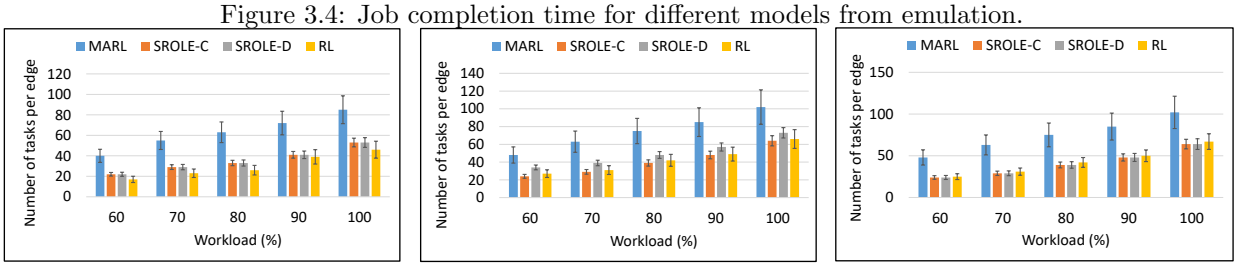
```
for each timestep do
    Collect actions from all edge nodes in the cluster  Virtually take the actions to assign layers to edges  foreach
    each edge node d_j do
        Calculate resource utilization u_k(d_j) of each resource
        Rank the assigned layers on d_j in descending order of resource demand weight  Punishment κ ← 0  while any
        u_k(d_j) > α, k = 1, 2, ... do
            Choose the top layer which is in action a_t  ã_t ← safe action by the shield  Replace a_t by ã_t  κ ← κ+
            constant negative reward  Notify the layer's scheduling edge about κ and ã_t
        end
    end
end
```

Algorithm 1 illustrates the pseudocode for the centralized shielding. At each timestep t (Line 1), the central shield observes the joint action and joint state of all the agents or edges in the cluster (Lines 2-3). For each edge node d_j, it calculates its resource utilization of each resource type (Line 5), ranks the assigned layers on d_j in descending order of resource demand weight (Line 6), and initializes κ (Line 7). It then checks whether its resource utilization is greater than the pre-defined threshold α (Line 8). If yes, the shield picks up the layer on the list top, finds and suggests a safe action for scheduling the layer and notifies the layer scheduling



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.4: Job completion time for different models from emulation.



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.5: The number of tasks per device for different models from emulation.

edge about the new action and the κ reward for the unsafe action (Lines 9-13). The shield repeats this process until the edge node will not be overloaded after it hosts all layers assigned to it (Lines 8-14).

3.4.4 Decentralized Shielding for MARL

A shield in a cluster is responsible for all edge nodes in a cluster and may become overloaded due to the communication and computation overhead in shielding. Thus, we propose a decentralized shielding method. In the decentralized shielding method, we first divide a cluster to multiple sub-clusters and each sub-cluster consists of geographically proximity-close edge nodes. Then, one shield works for one sub-cluster. Within each sub-cluster, the shield works in the similar way as described in the centralized shielding. There is one additional problem we need to handle. The edge nodes in the boundary of two or more sub-clusters may assign tasks to the same edge node in one sub-cluster, which may overload it, but the shield in this sub-cluster will not receive the actions from its neighboring sub-clusters and hence will not detect the action collision. To solve this problem, the shields of neighboring sub-clusters need to communicate with each other to avoid such a case. Specifically, the neighboring shields first select a delegate to check the action collisions. Then, they send the actions of the edge nodes and the available resources as well as the resource utilizations of the edge nodes in the boundary to the delegate. The delegate uses the same method in the centralized shielding method to check action collision and finds alternative actions. It sends the alternative actions to the neighboring shields, and the shields then forward the alternative actions to the corresponding edge nodes. As a result, the edge nodes take alternative actions instead of previously determined actions.

Table 3.1: Resource configuration for SROLE experiment.

Environment	Resource ranges
Real edge	Mem$\in$ $\{1024, 2048, 4096\}$MB CPU$\in$ $\{0.25, 0.5, 1.0\}$Host Ratio BW$\in$ $\{20, 100\}$MBps
Container	Mem$\in$ $\{768, 1024, 1536, 2048, 4096\}$MB CPU$\in$ $[0.3, 1.0]$Host Ratio BW$\in$ $\{50, 100, 200, 500, 1000\}$Mbps

3.5 Performance Evaluation

3.5.1 Experiment Setup

Emulation. Our proposed system runs on Tensorflow for the execution of the model training through its parameter server strategy. In order to emulate edges with varying resources, we use 25 docker containers. Each cluster has 5 edge nodes. The resource settings of our emulation and real device experiments are indicated in Table 3.1 and the resources of the devices were assigned in a round-robin way. The containers are deployed in Amazon EC2 instance of type m5ad.4xlarge. The CPU and memory are configured using commands from docker and the bandwidth between different containers is configured using the tcconfig tool [110].

Real experiments. Our real testbed consists of 10 Raspberry Pis; two Pis have 1 GB memory, four other Pis have 2 GB memory and four other Pi has 4 GB memory. They roughly have same CPU but we use the cpulimit [22] command to control the CPU as desired according to Table 3.1. The edges are connected via 2.4 GHz band wireless connection. We use wondershaper [120] tool to control bandwidth among the edges.

ML models and datasets. We run three ML models: GoogleNet Inception, VGG-16, and RNN [12]. We use the MNIST [53] dataset to run the first two models and the Air Quality dataset [115] for the RNN model. The MNIST dataset consists of 70,000 images of handwritten digits and is widely used for training CNN models. The Air Quality dataset contains 9358 instances of hourly averaged responses from an array of 5 metal oxide chemical sensors. One instance refers to the sensor values (as the ML inputs) and the AQI value (as the ML output). Since there are maximally 5 clusters, We divide the dataset to 5 subsets. Each cluster has a subset as the input training data and the data is randomly distributed among the edges in the cluster as their sensed data. We run three DL training jobs of the same type in each cluster initiated by randomly chosen edge nodes.

RL Training. To train the RL models, we need data related to both DNN models and edge nodes. To generate the data related different DNN model structures and we profiled and obtained their resource demands. To generate edge node configuration data, we consider the number of edge nodes in the range of [2,10]. For

each edge node, CPU is chosen randomly from range [0.5,2] GHz, memory is randomly chosen from range [64,4096] MB [30] and the bandwidth across pair of edge nodes is randomly chosen from range [128,1000] MBps [96]. Using these data, we train the RL model offline.

Workload and Settings. In all the cases, we trained one DNN model in each cluster and add several other non-ML jobs (PageRank [44]) from the HiBench benchmark to vary available resources on the edges. The workloads were controlled by running multiple PageRank job on these edges in a distributed way. We run x=2,3,...,6 PageRank jobs in each cluster throughout the whole training period to control the workload. Workload of 100% means there are 6 PageRank jobs running simultaneously in the system, and other workloads are defined similarly in the decreasing order, i.e., 5 is 90%, 4 is 80%, and so on. These jobs were run simultaneously with DNN training jobs until the run completes. During these experiments, we either change the number of edge devices or the workload along the x-axis. Unless otherwise indicated, the number of edge nodes is 25 and the workload is 100%. Each run for DNN model experiment was executed for 50 iterations. We repeated each experiment 5 times and plotted the median with the 5th and 95th percentile error bars. During the experiment, we measured the resource utilization of the devices every 10 minutes. In both the cases of emulation and real device, we set the value of the parameters $\alpha = 0.9$, $\rho = 1$, $\gamma = 50$ and $\kappa = -100$.

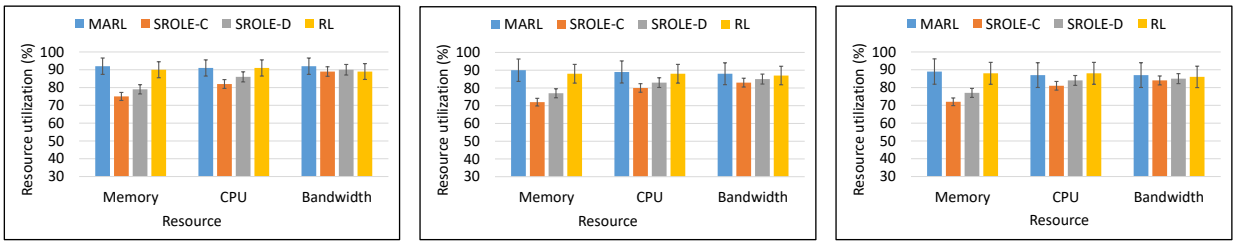
3.5.2 Compared Methods

MARL. This is a simple multi-agent RL-based method CQ-learning [42] without shielding. In particular, both the evolution of the system state and the reward received by each agent are influenced by the joint actions of all agents. That is, each agent has its own long-term reward to optimize, which now becomes a function of the policies of all other agents.

SROLE-C. This is a multi-agent RL method with an extra centralized shield. In the centralized shielding, there is a single shield to monitor all agents' joint actions and correct any unsafe action if necessary. That is, the shield observes all the agents and prevents any edge node from being overloaded.

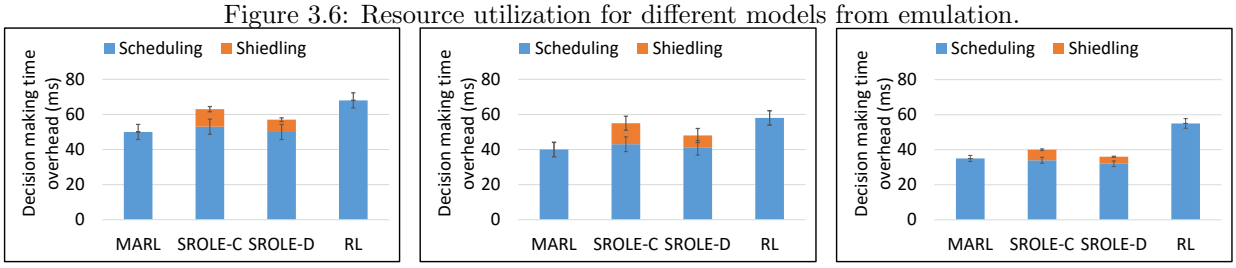
SROLE-D. This is an extension of the centralized shielding. It has multiple shields on multiple sub-clusters in a cluster and each shield is only responsible for the agents in its sub-cluster or a subset of agents in its cluster.

Centralized RL. In the following figures, we use *RL* for simplicity. This is an RL scheme where the cluster head makes the assignment decision for all the jobs in its cluster. In this method, we assign a negative reward



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.6: Resource utilization for different models from emulation.



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.7: Computation overhead for different models from emulation.

for overloading the memory of a certain device and otherwise, the reward is based on the job completion time.

3.5.3 Metrics

Job completion time. This is the training time, which denotes the time period from the time when a job starts to run after scheduling to the time when the training of the whole model completes. This time period contains multiple number of iterations depending on the size of the dataset. In our case, the training for all the models comprises of 50 iterations.

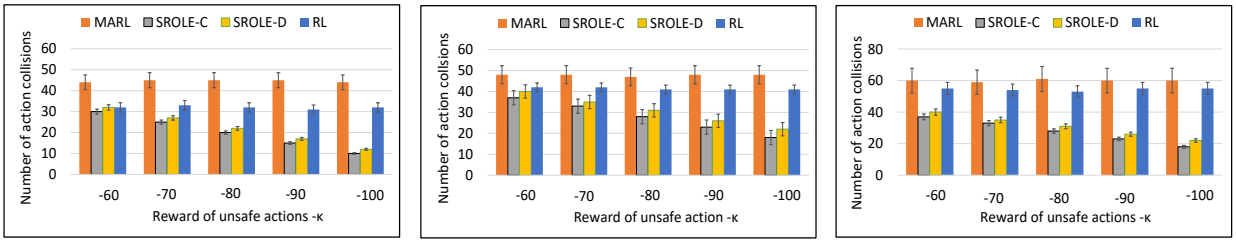
The number of tasks per device. From different runs, we measure the number of partitions for a DNN training job and tasks for non-ML jobs running on each device. This measurement is to show the performance of avoiding overloading edge nodes.

Computation time overhead. Computation overhead refers to the decision making time of each method. It is the time period from the time when a job is initiated to the time when the task assignment schedule of the job is made.

The number of action collisions. We measure the number of action collisions for the negative reward (κ) for unsafe actions.

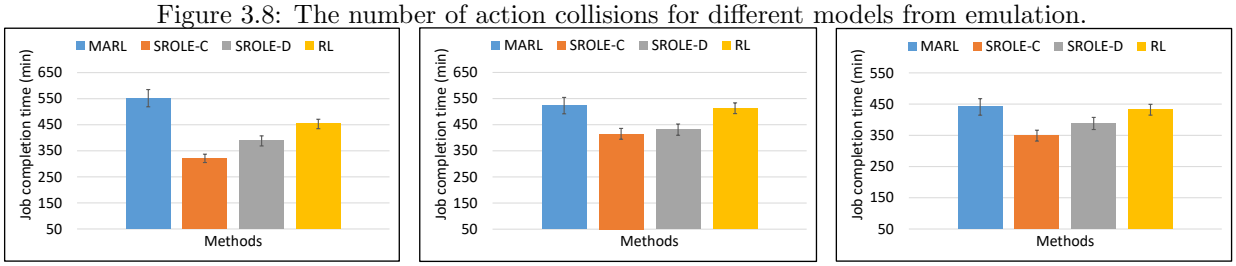
3.5.4 Experimental Results from Emulation

Job completion time. Figures 3.4a, 3.4b, and 3.4c show the job completion time versus the number of edges for training the VGG-16, GoogleNet, and RNN models, respectively. MARL and RL have similar job



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.8: The number of action collisions for different models from emulation.

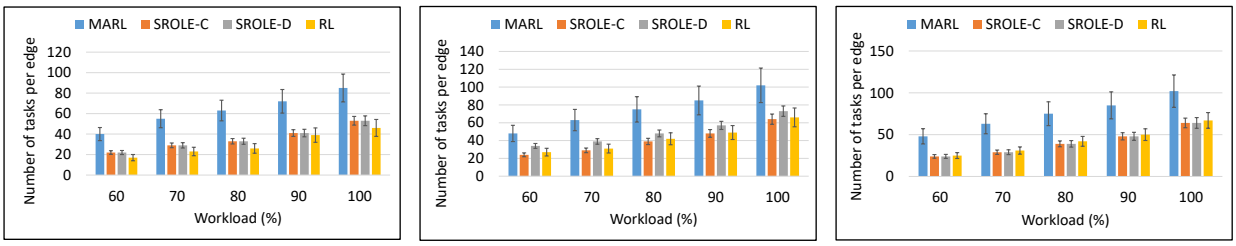


(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.9: Job completion time for different models from a real-device network.

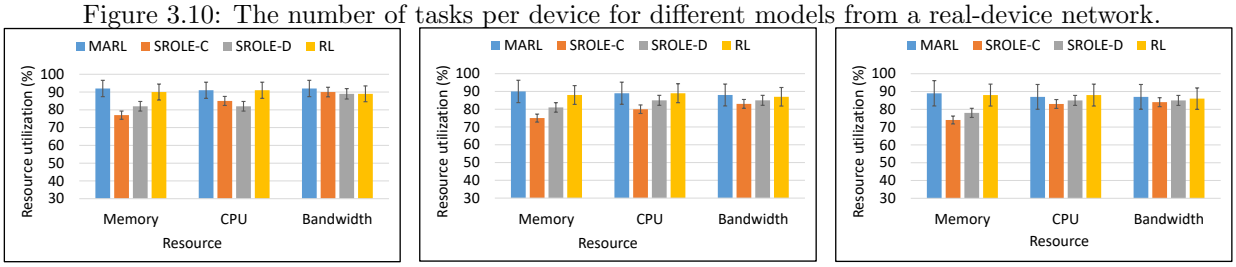
completion times, which indicates that the performances of their job schedules are similar. The results imply that MARL still can achieve comparable performance as RL though MARL does not have global information for job scheduling. Both centralized and decentralized shielding methods (SROLE-C and SROLE-D) perform better than RL and MARL without shielding because shielding reduces the number of unsafe actions, i.e., overloading individual devices. As a result, it reduces the job completion time. For VGG-16, GoogleNet, and RNN, SROLE-D shows 36-45%, 35-43%, and 33-44% reduction in job completion time than MARL or RL without shielding. SROLE-D performs 8-13% less than SROLE-C for all three models because the action collisions are checked by multiple shields instead of one, adding extra communication time among the neighboring shields during the DNN training. As a result, SROLE-C saves job completion time by 49-56% for VGG-16, 48-59% for GoogleNet, 47-56% for RNN, respectively in comparison with RL and MARL without shielding. Thus, the shielding can be conducted more efficiently because of observing the resource state of all the edges together. From all the figures, we see that as the number of edges increases, the job completion time increases. This happens because more clusters lead to more time in transferring the model parameters from the clusters to the parameter server for synchronization of the model. Also, the figures show the results do not vary greatly and keep relatively stable.

The number of tasks per edge node. Figures 3.5a, 3.5b, and 3.5c show the number of tasks per node versus different workloads for training the VGG-16, GoogleNet, and RNN models, respectively in the scenario with 25 edges. We plot the median (denoted with different colors) along with the minimum and the maximum number of tasks (denoted with black bars). SROLE-D shows 42-56%, 46-61%, and 41-56% reduction in the median number of assigned tasks per device for VGG-16, GoogleNet, and RNN compared to MARL or RL



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.10: The number of tasks per device for different models from a real-device network.



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.11: Resource utilization for different models from a real-device network.

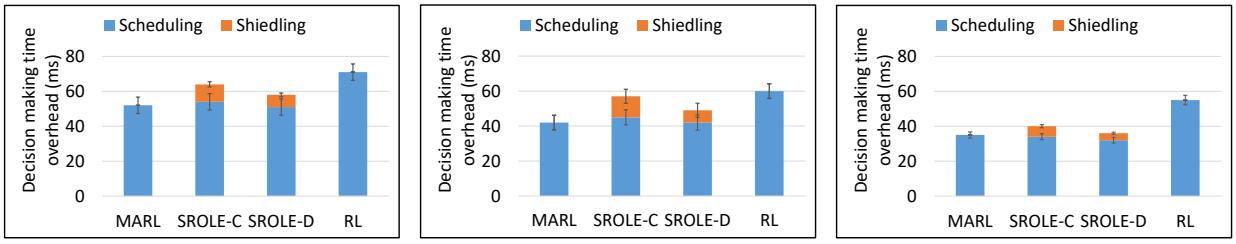
without shielding, respectively. However, the SROLE-C outperforms the SROLE-D by 2-11%. As a result, SROLE-C generates a reduction in the median number of assigned tasks per device by 49-56% for VGG-16, 48-59% for GoogleNet, 47-56% for RNN respectively in comparison with RL and MARL without shielding. From the figure, we also observe that both the SROLE-D and SROLE-C methods show less variance than both MARL and RL without shielding. Both SROLE-C and SROLE-D perform better than other methods because shielding reduces the number of unsafe actions, i.e., overloading on individual devices, and thus distributes the tasks among devices in a more balanced manner. However, SROLE-D performs less than SROLE-C shielding because of taking a higher number of unsafe actions. This phenomenon happens because of not knowing the cluster more completely.

Resource utilization. Figures 3.6a, 3.6b, and 3.6c show the resource utilization of each type of resources for training the VGG-16, GoogleNet, and RNN models, respectively in the scenario of total 25 edges. We plot the median along with the minimum and maximum resource utilizations as error bars. For VGG-16, GoogleNet, and RNN, SROLE-D shows 12-19%, 11-17%, and 11-15% reduction in the median resource utilization compared to MARL or RL without shielding. However, the SROLE-C outperforms the SROLE-D by 2-14%. As a result, SROLE-C generates a reduction in the median resource utilization by 21-24% % for VGG-16, 48-59% for GoogleNet, 22-29% for RNN, respectively in comparison with RL and MARL without shielding. From the figure, we also observe that both the SROLE-D and SROLE-C show less variance than both the MARL and RL without shielding in terms of resource utilization. Both centralized and decentralized shielding methods perform better than other methods because shielding avoids overloading individual devices. However, SROLE-D performs less than SROLE-C because of taking a higher number of unsafe actions.

This happens due to not having the complete knowledge of the cluster, which adds extra computation or communication for the involvement of multiple shields.

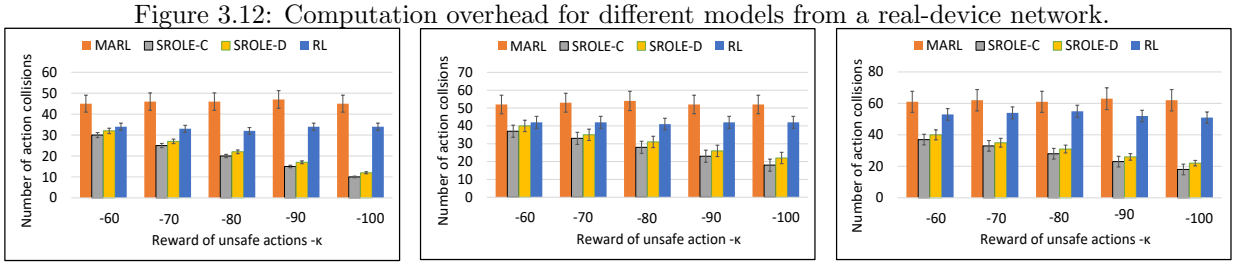
Average computation time overhead. Figures 3.7a, 3.7b, and 3.7c show the computation overhead for scheduling (blue bar) and shielding (orange bar) of different methods while training the VGG-16, GoogleNet, and RNN models, respectively. For all the models, the results for computation overhead (scheduling + shielding) are as follow: MARL<SROLE-D<SROLE-C<RL. RL needs the longest decision making time because only one node is responsible for scheduling all jobs in one cluster. MARL greatly reduces the decision making time of RL since MARL distributes the scheduling load among the edge nodes in the cluster by letting each edge node schedule its own job among its neighbors. Both RL and MARL do not have any shielding time as they do not have the shielding approach. SROLE-C and SROLE-D are based on MARL, and they have additional shielding components to detect action collisions, thus generating higher decision making time. MARL, SROLE-C and SROLE-D have the same scheduling time since they all use MARL for scheduling. SROLE-D generates 5-8% less shielding time than SROLE-C for all the models. This is because SROLE-C relies on one shield in each cluster, so the shield needs to check all the actions, which needs a long time, and SROLE-D distributes the shielding overload among multiple shields, which expedites the shielding process and reduces both the shielding time and the decision making time.

The number of action collisions. Figure 3.8 shows how the assigned reward of unsafe action impacts the number of unsafe actions during the training of all the DNN models. SROLE-C performs 31-48% better than the MARL or RL, while SROLE-D performs 27-39% better than MARL or RL for all the three models. This is because the added shield(s) in SROLE-C and SROLE-D coordinate the edges to avoid unsafe actions, but MARL and RL do not have shields. The SROLE-C approach performs 4-7% better than SROLE-D. This is because the global shield in SROLE-C can observe the global environment, compare all actions and suggest alternative actions accordingly to avoid unsafe actions globally. When multiple shields are responsible for sub-clusters in SROLE-D, the information collected by a shield for the boundary nodes may not cover all the unsafe actions, leading to unsafe actions. Though SROLE-C and SROLE-D use the shielding approach, they still produce certain unsafe actions. This is because the resource demands of tasks are time-varying and dynamic and sometimes cannot be accurately predicted, thus leading to the edge node overload. For all the three models, as the absolute value of the reward of an unsafe action increases, the number of unsafe actions during the whole training period in SROLE-C and SROLE-D decreases and this number in MARL and RL keeps constant. MARL and RL do not use this reward or shielding approach, so their performances are not affected by the reward value. A high penalty for the action collision will help SROLE-C and SROLE-D avoid more unsafe actions.



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.12: Computation overhead for different models from a real-device network.



(a) VGG-16. (b) GoogleNet. (c) RNN.

Figure 3.13: The number of action collisions for different models from a real-device network.

3.5.5 Experimental Results from a Real Device Network

We formed the 10 edge nodes into a network and considered it as a single cluster, and then ran real experiments on the real-device network. From the real experiments, we observe similar performances of the different methods as in the container-based emulation due to the same reasons mentioned above. **Job completion time.** Figure 3.9 shows the the job completion time for training all models in the real-device cluster. For these models, SROLE-D performs 32-39% better than MARL or RL without shielding and SROLE-C performs 36-53% better than MARL or RL. SROLE-D performs 4-7% worse than SROLE-C because SROLE-D has additional communication operations between neighboring shields for shielding.

The number of tasks per edge. Figure 3.10 shows the number of tasks per node for training all the models. Comparing to MARL and RL without shielding, SROLE-D shows 28-45% reduction and SROLE-C shows 39-52% reduction in the median number of assigned tasks per device. SROLE-D performs 7-11% worse than SROLE-C because the shields in SROLE-D do not have global knowledge of the state of the resources in all edges. Similar to the emulation experiments, the variances of SROLE-D and SROLE-C are lower than those of MARL and RL without shielding for all the models.

Resource utilization. Figure 3.11 shows the resource utilization of each type of resources for training the three models. SROLE-D shows 18-23% reduction and SROLE-C shows 21-28% reduction in the median in the median of resource utilization. SROLE-D performs 3-5% worse than SROLE-C because the shields in SROLE-D do not have global knowledge of the state of the resources in all edges. Similar to the emulation

experiments, the variances of SROLE-D and SROLE-C are lower than those of MARL and RL without shielding for all models.

Average computation time overhead. Figure 3.12 shows the computation overhead for scheduling and shielding of different methods while training all the models. For the shielding time, SROLE-D performs 4-7% better than SROLE-C for real-devices.

The number of action collisions. Figure 3.13 shows how the assigned reward of unsafe action impacts the number of unsafe actions during training the DNN models. SROLE-D shows 27-42% reduction and SROLE-C shows 29-46% reduction in the median in the median of resource utilization. SROLE-D performs 2-6% worse than SROLE-C because the shields in SROLE-D do not have global knowledge of the state of the resources in all edges.

3.6 Conclusion

Fully distributed DNN training on edges utilizing concurrent model and data parallelism is a promising way to increase the scalability of DNN model training at resource-constrained edges. However, relying on one edge node to use RL to schedule the model partitions (i.e., distributing the partitions of a large DL model to a set of edges to minimize the training time) among edge nodes is not scalable. In this paper, we propose a multi-agent RL system that enables each edge node to schedule its own jobs. To ensure such distributed job scheduling method will not overload an edge node, we propose using shielding that observes the actions decided by all edge nodes to avoid overloading edge nodes. Relying on one shield is not scalable. Thus, we further propose a decentralized shielding method that relies on multiple shields to conduct shielding in a distributed manner. Our experiments show that our shielding method performs 59% better than multi-agent RL in training time with 29% less median resource utilization of an edge device, and also the multi-agent RL method achieves similar job completion time performance as the centralized RL method. In the future, we will explore using formal method approaches to provide a guarantee of action collision avoidance and explore a method to avoid action collisions caused by decentralized shielding.

Chapter 4

Input aware Layer Assignment between CPU and LCA during Inference

The chapter handles the challenge highlighted in Chapter 1: Variation of Performance among the Edge Accelerators. The chapter investigate this issue from the perspective of different accelerators such as GPU, and TPU. Previous work pre-determines a model's layer assignment between the CPU and accelerator offline for high accuracy and low latency without considering the input. However, we observe that the input affects the optimal layer assignment. To address this problem, in this paper, we present a system for Fast, Accurate DNN Inference on Low-Cost Edges using Heterogeneous Accelerator eXecution (FLEX). By leveraging observations common to the models deployed at a variety of edge devices, we design FLEX that has a lightweight heuristic and an RL-based mechanism to dynamically determine the layer assignment of a model across the CPU and the accelerator given an input.

4.1 Introduction

In the recent past, Machine Learning (ML) models have been extensively used for various applications in our daily lives ranging from transportation [19] and image/video processing [13] to healthcare applications such as detecting user action [105], emotions [51], abnormalities in vitals [9, 23] and infections [153]. A principal

This chapter is selected for publication Proceedings of EuroSys'25, titled " Flex: Fast, Accurate DNN Inference on Low-Cost Edges Using Heterogeneous Accelerator Execution".

reason behind the success of such technological breakthroughs is the ubiquity of the large number and the wide variety of sensors [91, 128] in edge devices today–smartphones, tablets and smartwatches–that generate inputs to these ML models. The ubiquitous availability of such rich data has enabled ML models to achieve or even exceed human performance [34].

Compared to the cloud [36], the advantages of *on-device* ML in terms of latency, privacy, and network connectivity have resulted in edge devices becoming powerful enough to execute even complex ML models locally [101]. Modern edge devices, including flagship Android and Apple smartphones and tablets, are equipped with GPUs that can greatly accelerate ML tasks [32]. Modern accelerators often employ higher precision arithmetic, which leads to increased energy consumption. Consequently, applications like AWS SageMaker and Elastic Inference allow users to configure precision settings to optimize energy usage [152], thus transforming modern accelerators into low-cost accelerators (LCAs).

In addition, a significant portion of edge device users resides in regions where these devices are prohibitively expensive. For instance, as of January 2023, approximately 70% of smartphone users in Africa, Asia, and South America use low-cost Android devices [35]. These budget-friendly devices typically feature LCAs like NPUs, TPUs, or DSPs (e.g., Qualcomm Snapdragon with Hexagon AI engine). Despite their cost-effectiveness, these cheaper accelerators compromise on accuracy due to the use of low-precision arithmetic [20].

In this paper, we focus on improving the accuracy and time efficiency of model inference on low-cost edge devices equipped with LCAs. The closest related work to our work is Machine Learning based Model Partition algorithm (MLMP) [107] which pre-schedules a model between the CPU and accelerator offline without considering the input. However, we observe that the input affects the optimal schedule so such input-agnostic methods are non-optimal. In addition, MLMP doesn't consider assigning the model layer-by-layer, but instead utilizes a sorting mechanism or enumerates a number of layer assignments (based on sorting) to optimize for either accuracy or latency greedily. However, these layer assignments may not include better solutions may exist. To address these problems, we present a system for Fast, Accurate DNN Inference on Low-Cost Edges Using Heterogeneous Accelerator eXecution (FLEX) that dynamically splits and executes a model between a device's CPU and LCA for a given input. FLEX is based on a key observation that while LCAs give up accuracy, *individual layers of a model are affected disproportionately* regarding the accuracy loss and the execution speedup in such accelerators. Accordingly, FLEX places layers that are unlikely to lose accuracy but will be executed faster in the LCA and the rest of the layers in the CPU.

FLEX is designed by leveraging observations common to the models deployed at a variety of edge devices. It incorporates several novel techniques to realize its goals. First, it has a lightweight ML model to predict, at a

layer granularity, whether executing the layer *with the given input* will result in the same answer in the CPU and the LCA. Once Flex confirms the per-layer output matches through profiling, it solves an optimization problem that determines the optimal placement of layers. Initially, Flex employs a heuristic approach to address this problem. Subsequently, it transitions to a reinforcement learning (RL) method once the RL model is fully trained using data from the heuristic method.

Our real device experiments show that Flex achieves improvement of up to 36% in average inference time, up to 20% in average accuracy, up to 35% in timeliness guarantee ratio, up to 31% in accuracy guarantee ratio, and 58% in consumed energy compared to the state-of-the-art approaches. Flex's source code is open-sourced [127].

We make the following contributions in this paper:

- We conduct an evaluation of low-cost accelerators for ML inference on different edge devices, and show that a static layer assignment of a model would not entail useful accuracy guarantees. Thus, we make the case for an input-specific partitioning of a model. (sec:motivation)
- We make several novel observations that enable us to design Flex, a lightweight and practical execution framework that dynamically finds the layer assignment for optimal latency and accuracy. To the best of our knowledge, Flex is the first technique that can support real-time, per-input dynamic partitioning of a model across the CPU and LCA for latency and accuracy efficient execution. (sec:motivation, sec:damie)
- We conduct an extensive evaluation of Flex and show that not only does it significantly outperform the state-of-the-art, but it also achieves accuracy results only 6% less optimal compared to the maximum achievable results. (sec:exp)

4.2 Motivation and Foundation of Design

In this section, we first discuss the advantages and shortcomings of LCAs in the light of a typical object or person identification recognition application. Such applications typically come with a specific timeliness and accuracy guarantee [136]. Then, we describe why statically partitioning a model is not enough, and that incorporating the input for such an application as part of the partitioning process is necessary to achieve accuracy targets. Then, we present the observations that enable us to arrive at a solution and the challenges in realizing the solution.

Experiment settings. We used an Android smartphone equipped with Snapdragon 778G 5G containing a Cortex-A78 4-core CPU with 2.4 GHz clock speed and an Adreno 642L 4-core GPU along with 128GB

Table 4.1: Models and datasets used.

Model	Task	Dataset
VGG-16	Vision (Image classification)	ImageNet [25]
VGG-19		
ResNet-34		
ResNet-50		
MobileNetV3	Vision (Object recognition)	
YoloV7-tiny		
MobileBERT	NLP (Question answering)	SQuAD [92]
LSTM	Time-series (Human activity recognition)	HAR [5]

storage and a Hexagon 770 AI engine. We also used a Google Pixel 6 that comes with a 2-core TPU to show that our observations hold across LCAs. Unless otherwise mentioned, we used the models and the datasets listed in Table 4.1, and used the same settings described in the following. We ran 100 requests for each model. As in [116], we used the Poisson distribution of mean 3 seconds to simulate the arrivals of inputs. Throughput this paper, the figures with error bars plot the median values along with the 1^{st} and the 99^{th} percentiles.

4.2.1 Low Cost Accelerators: Pros & Cons

We conducted an experiment to illustrate the usefulness and issues with LCAs. We ran the models in two settings: entirely on the CPU and entirely on the LCA. For LCA, we measured the metrics both in GPU and TPU and showed the average of the results in Figure 4.1. As we see, executing the model entirely on the LCA significantly speeds up the inference tasks, but loses up to 7.3% accuracy compared to what is achievable. Such large accuracy drops may not be acceptable in practice.

4.2.2 Static Partitioning: Shortcomings

MLMP [107] partitions a model execution between the CPU and the LCA in a *static* fashion. It aims to maximize accuracy while satisfying the time requirement (denoted by MLMP-A) or minimize the inference time while satisfying the accuracy requirement (denoted by MLMP-T). It uses a regression model to predict the final accuracy and inference time of the model for a specific layer assignment. MLMP-A assigns the model to the LCA first and then greedily moves layers to the CPU until the time requirement is not satisfied, while MLMP-T assigns the model to the CPU first and then greedily moves layers to the LCA until the accuracy requirement is not satisfied. However, it doesn't take into account the effect of the input data on the partitioning decision.

By enumerating all layer assignment schedules, we found the optimum layer assignment (called *Oracle*) that achieves the accuracy guarantee (i.e., the same output as the one when all layers ran in the CPU) and the minimum inference time. We compared the performance of *Oracle* for different models with three other methods: MLMP-A, MLMP-T, and a simple input-aware layer assignment method, denoted by *Input-aware*.

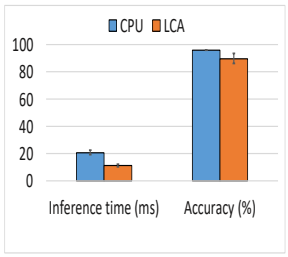


Figure 4.1: LCAs acceler-ate models at the cost of ac-curacy loss.

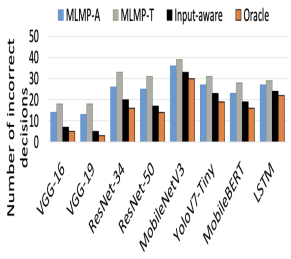
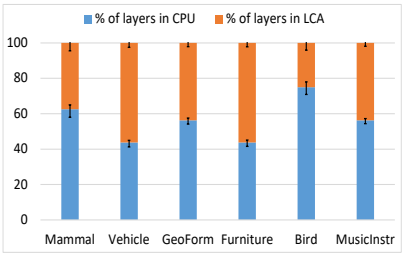
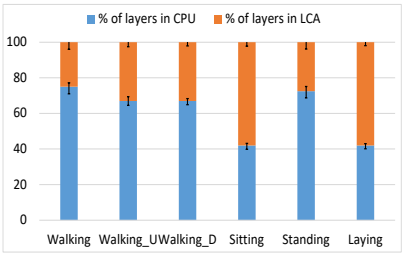


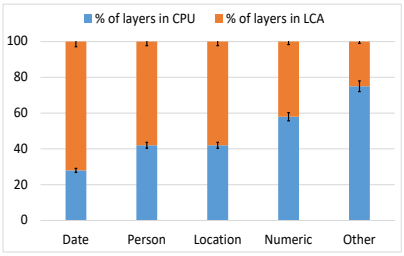
Figure 4.2: Incorrect deci-sions compared to *Oracle*.



(a) Imagenet.

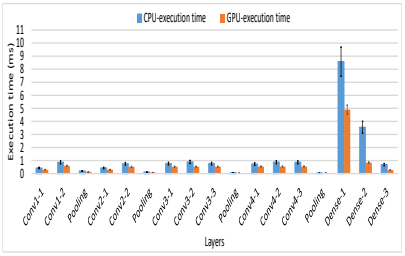


(b) HAR.

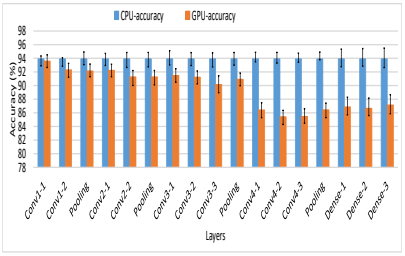


(c) SqUAD.

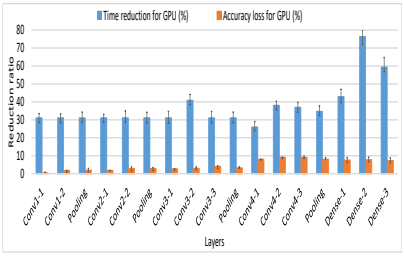
Figure 4.3: Percentage of layers in CPU and in LCA in *Oracle*.



(a) Execution time.

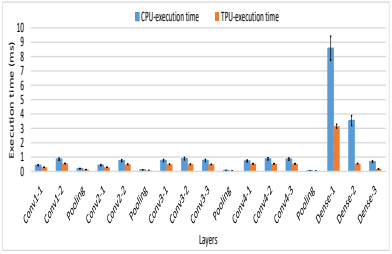


(b) Accuracy .

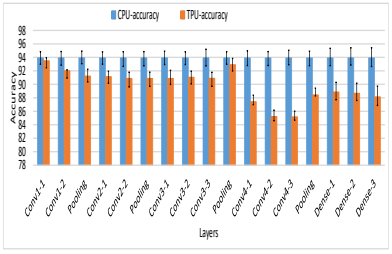


(c) Reduction ratio per input.

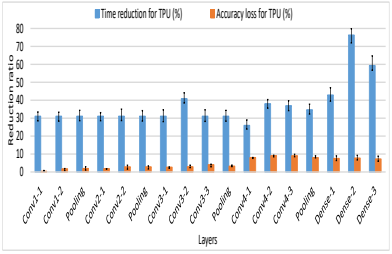
Figure 4.4: Performance difference in CPU and GPU for VGG-16.



(a) Execution time.



(b) Accuracy.



(c) Reduction ratio.

Figure 4.5: Performance difference in CPU and TPU for VGG-16.

In *Input-aware*, we keep moving the layers to the GPU one by one from the end of the DNN model until we reach a point where the accuracy guarantee is violated. During this process, we measure the inference

time and accuracy each time. From those assignments, we chose one at random. All these methods were implemented with the Tensorflow framework using Android NDK [4].

fig:misdecmodels shows the count of incorrect decisions made by each method when compared to *Oracle* for various models. We see that MLMP-A and MLMP-T make significantly more incorrect decisions, and their error rates are almost 2× higher than that of the *Input-aware* method. In contrast, *Input-aware* has only ≈ 2.5% more incorrect decisions in comparison with *Oracle*. The major cause for the poor performance of MLMP techniques is the fact that it does a static partitioning of the model once and applies it across all inputs. To verify this, we conducted another experiment to show how the layer assignments vary between CPU and LCA for different categories of inputs. Figure 4.3 shows the percentage of layers assigned to the CPU and the LCA for different categories (labeled class) for different datasets with the accuracy requirement of 85% in *Oracle*. The figure shows that individual categories have different percentages of layers assigned between the CPU and the LCA, and within one category, the variance of the percentages is very low. The results imply the importance of input-awareness in determining the layer assignment.

Then, one may ask if the distribution of the inputs is similar to that of the training data, do we still need to consider the individual inputs? To answer this question, we ran KS-test [90] over the 100 inputs processed by each model to measure the drift of the inputs from the training data of the models. We observed that the p-values [90] for the ImageNet, HAR, and SQuAD datasets are 0.14, 0.09, and 0.18, respectively. These values are greater than the 5% significance level value (i.e., 0.05) [90], which means similar distributions. This outcome suggests that even though the inputs share a similar distribution as the training data, the input-awareness is still necessary.

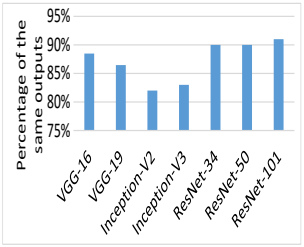
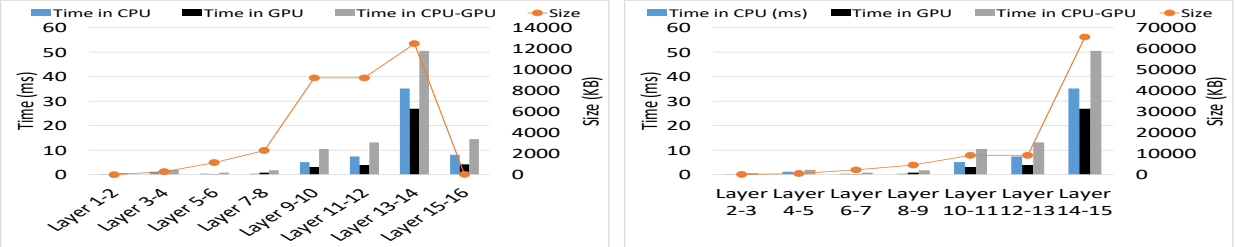


Figure 4.6: Percentage of the same outputs for the models.

Observation 1: To achieve the maximum accuracy, finding the right layer assignment between the CPU and accelerator of a DNN model for each input is necessary instead of using a static, one-time partitioning of the DNN model. Unfortunately, due to the exhaustive nature of the search required for arriving at a solution and the prohibitive computational requirements for doing so, solutions such as MLMP cannot support custom partitioning in real time for each input. In the following, we take the VGG-16 model as an example to show our analysis results.



(a) Pair-wise time for odd pairs.

(b) Pair-wise time for even pairs.

Figure 4.7: Performance for pairs in CPU and GPU for VGG-16.

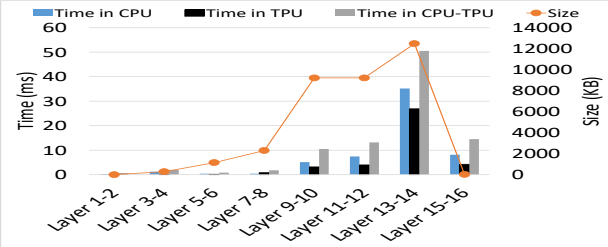
4.2.3 Effect of Layer on Latency & Accuracy

We ran the trained VGG-16 model on GPU device and TPU device for the ImageNet dataset using the TensorFlow lite. We measured the execution time and accuracy when all layers run on the CPU (denoted by CPU execution time and CPU accuracy), and when each layer runs on GPU and all other layers on CPU (denoted by GPU execution time and GPU accuracy), respectively. We ran the experiment 5 times, and each experiment was executed for 500 input samples. Figures 4.4 and 4.5 show these results and their reduction ratio when one layer runs on GPU in the GPU device and TPU device, respectively. From the experiment, we learn that even though the GPU or TPU speeds up the execution time for each layer compared to the CPU by 30% on average, only the latter few layers account for the bulk reduction of the execution time. The same for accuracy. Allocating the layers to GPU and TPU accounts for approximately 5% loss in accuracy compared to CPU on average. This means the outputs of running a layer in CPU and LCA (while other layers are on CPU) could match.

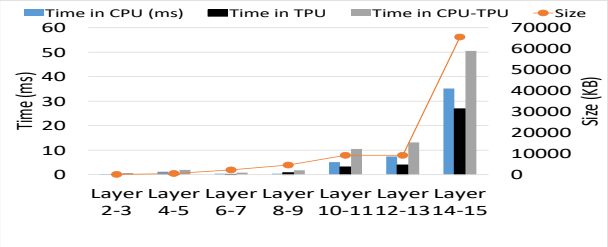
Observation 2: Different layers have different time and accuracy reduction ratios for the LCA in comparison to the CPU. Therefore, to choose layers to move from CPU to LCA, among the layers for which the outputs of running on CPU and LCA match, the layers that contribute to the objective metric the most should be chosen.

Figure 4.6 shows the percentage of the cases when the outputs match when each entire model ran on the CPU or the LCA. We plot the average of the percentage values from the GPU and TPU devices. We see some models have a high percentage of the same outputs.

Observation 3: The "heavy hitter" layers have higher reduction ratios in both inference time and accuracy when running on LCAs in comparison to the CPU, so when the outputs of running the model on CPU and LCA match, these layers should run on the LCA, and otherwise on the CPU. (fig:anc1,fig:anc10t,fig:accinf111)

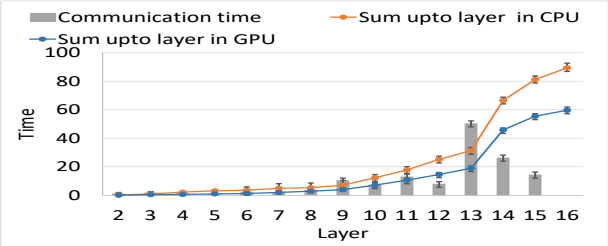


(a) Pair-wise time for odd pairs.

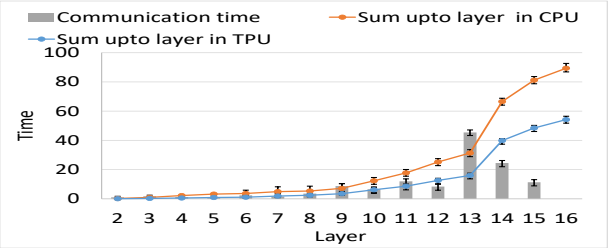


(b) Pair-wise time for even pairs.

Figure 4.8: Performance for pairs in CPU and TPU for VGG-16.

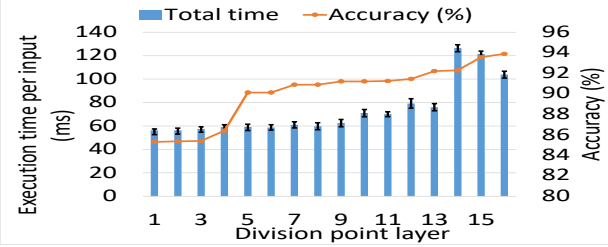


(a) GPU.

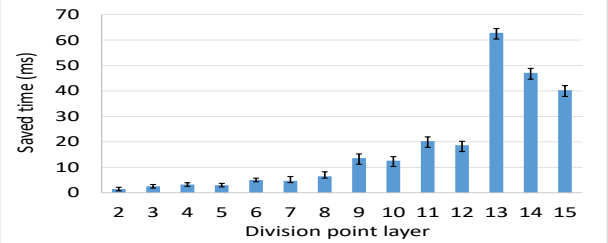


(b) TPU.

Figure 4.9: Time between CPU and LCA.

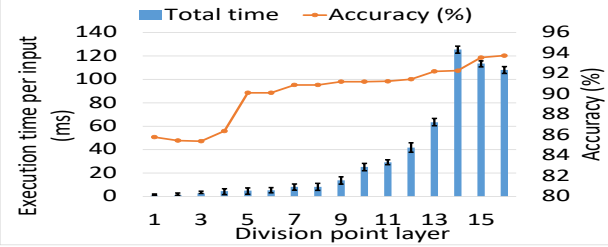


(a) Total time and accuracy.

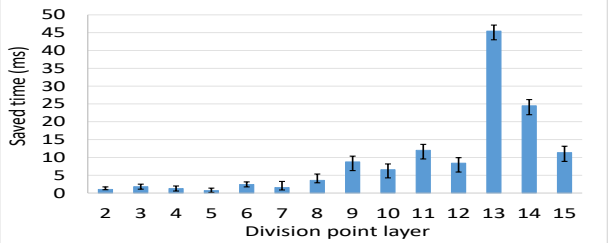


(b) Saved time.

Figure 4.10: Performance for different division points of VGG-16 in GPU.



(a) Total time and accuracy.



(b) Saved time.

Figure 4.11: Performance for different division points of VGG-16 in TPU.

4.2.4 Effect of Data Movement between CPU and LCA

As the communication between the CPU and the LCA generates time latency, we measure the time needed for running each pair of consecutive layers of the VGG-16 model when they are all on CPU, all on LCA, or across CPU and LCA, as well as the size of the embeddings or the data parameters transferred between

the consecutive layers. The results for the GPU device and the TPU are shown in Figures 4.7 and 4.8, respectively. Among the three possible placements, the execution time of the placement across CPU and LCA is the largest, and it is proportionate to the size of the parameters, while the placement only on the GPU is the smallest.

Determining the division point in the model across the CPU and LCA needs to consider the CPU-LCA communication time. We then measure the execution time and communication time when we divide the model at each layer and show the results for the GPU device in Figure 4.9. In the figure, the "sum upto layer in CPU" when x=5 means that the layers from 1 to 5 are placed on CPU, and the rest of the layers are placed on the LCA, and the same concept is applied to the GPU. The communication time for that layer means the time to transfer data from layers 5 to 6. From the figure, we see the layers from the last have higher computation time in both CPU and GPU. There are certain divisions in a DNN model, for which the data transfer time between the consecutive layers is higher than the computation time of the predecessor layers assigned in either CPU or GPU.

We chose different division point layers, and ran this layer and its preceding layers on the CPU, and ran its succeeding layers on the GPU. Figures 4.10a and 4.11a show the inference time and accuracy for each division point layer for each input. We observe that as we move more layers from GPU to CPU, the inference time increases, and the accuracy also keeps increasing. Figures 4.10b and 4.11b show the saved time compared to running all the layers in the CPU per input at each division point. We use T^{CPU} as the total time if all layers are executed in CPU, and T_{κ}^{LCA} as the total time if the layers from layer $\kappa + 1$ to the last layer are executed in LCA. Let's use T_{comm} to denote the communication time between the layer at division point κ and its next layer. Then, the saved time is calculated by $T_s = T^{CPU} - T_{\kappa}^{LCA}$-$T_{comm}$. The figure shows that the saved time usually keeps increasing as we move the division point from left to right. However, there are a few specific ranges (e.g., 10-11, 12-13, 14-16) where it decreases instead of increasing. .

Observation 4: Generally, the accuracy and saved time for the LCA in comparison to the CPU increase as the division point moves from the first layer to the last layer in spite of some exceptions, but the communication time could be higher than the computation time. Therefore, it is necessary to pick the division point layer based on the trade-off between inference time (including the communication time) and accuracy.

4.2.5 Solution Overview

Enabling a per-input assignment of a model across CPU and LCA is a challenging task. FLEX leverages the observations we made in this section to build towards a solution. First, for a model, it is empirical to choose

the right layer assignment for each input, which is necessary as per *Observation 1.* As a result, the solution approach to find the layer assignment between CPU and LCAs should be as lightweight as possible. Later, as per *Observation 2*, FLEX moves layers from CPU to LCA based on whether the outputs from CPU and LCA match for a layer and its contribution to the objective metric. Also, as per *Observation 3*, since the heavy-hitter layers have a higher reduction ratio than the shallow layers, FLEX focuses on the heavy-hitter layers for the assignment on the LCA based on whether the outputs of the CPU and LCA match to each other. Finally, there is a time-accuracy trade-off between the CPU and LCA depending on the division point layer. As per *Observation 4*, while assigning the layers between the CPU and LCA, data transfer time also plays a crucial role, as it is usually higher than the computation time of all the predecessor layers at the division point. Consequently, FLEX focuses on dividing the layers at the point that has a lower communication time, which is important to meet the goals.

4.3 Flex Design

The architecture of FLEX is depicted in fig:sys. FLEX has offline-trained models based on profiled data. After it receives an input ①, it determines the layer assignment ② with assistance from the offline-trained models. Then, it loads the layers to CPU and LCA accordingly ③, and runs the model to output the result ④. FLEX consists of the following components described below.

- **Offline Profiling.** FLEX first profiles different layer assignments for a DNN model using a device's resource configuration and different inputs. Using profiling, FLEX predicts both the inference time, accuracy and whether the CPU result and LCA result will match. The prediction model is used in the heuristic and RL methods below.
- **Heuristic Methods.** For a model, FLEX chooses the layer assignment that meets the deadline and accuracy requirements for each input based on the layer-wise reduction of accuracy and inference time for an input. Furthermore, to reduce the overhead for the layer-wise assignment for each input, FLEX has an alternative solution that decides one division point of the DNN model between the CPU and LCA.
- **RL-based Method.** FLEX also has an RL-based approach to further reduce the decision-making time. It automatically outputs the solution for each input. Curriculum learning is used for the generalization of the RL-based approach for different models and faster convergence.

FLEX first runs the heuristic and uses the collected data for training the RL model, and then switches to use the RL-based method when the model is well trained from heuristic.

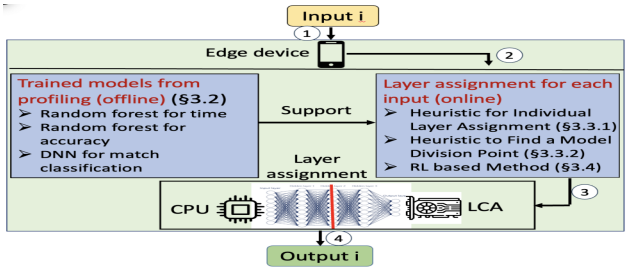


Figure 4.12: The architecture of FLEX.

4.3.1 Problem Formulation

The inputs of the problem contain a set of n inputs for an inference model j, $\mathbf{I} = \{I_1, I_2, \cdots, I_i, \cdots, I_n\}$ and user-specified requirements on the latency and accuracy for the application. Let $L^i = (x_1^i, x_2^i, \cdots, x_k^i, \cdots, x_K^i)$ denote a layer assignment for an inference model j for an input i, where $x_k^i \in \{0, 1\}$ is a binary variable, and $x_k^i = 1$ indicates that the k^{th} layer (l_k) runs on the LCA input, and $x_k^i = 0$ otherwise. Now, let $T(L^i)$, $A(L^i)$ and $M(L^i)$ denote the inference time, accuracy, and the used LCA memory of running the DNN model with a layer assignment L^i.

We use T_{goal}^i and A_{goal}^i to represent the deadline and accuracy requirements of input i, and use M_d to denote the total GPU memory of the device. We define the objective function as:

$$O^i = \alpha(A(L^i) - A_{goal}^i) + (1 - \alpha)(T_{goal}^i - T(L^i)) \tag{4.1}$$

The problem can be formulated as follows [31]:

$$\sum_{i=1}^{n} \arg\max \frac{O^i}{n} \tag{4.2}$$

subject to constraints:

$$\forall_i T(L^i) \leq T_{goal}^i \tag{4.3}$$

$$\forall_i A(L^i) \geq A_{goal}^i \tag{4.4}$$

$$\forall_i M(L^i) \leq M_d \tag{4.5}$$

Parameter $\alpha \in [0, 1]$ controls the importance between the accuracy and the time. Constraints (4.3) and (4.4) ensure that the chosen layer assignment for the model meets the latency and accuracy requirements for all the inputs. Constraint (4.5) ensures that the chosen layer assignment does not overflow the GPU memory for all inputs.

4.3.2 Offline Profiling

We use TensorFlow benchmark [112] to conduct offline profiling on the cloud to estimate the accuracy and inference time for different layer assignments L^i for each input for each model. We vary the structural parameters (e.g., input and output dimensions, kernel height, and kernel width [133]) of different types of layers within reasonable ranges as indicated in [133] to generate different models. We create the layer assignments adopting the approach mentioned in [107]. That is, we generate a random number p from $[1, K]$. Then, we schedule each layer to run on LCA with probability p/K. We profile the inference time and accuracy (i.e., confidence value).

Using the profiled data, we train two Random Forest regressors [11] for estimating inference time and accuracy, named *RF-T* and *RF-A*. The inputs include the structural parameters, the layer assignment and the input sample.

For each layer running on LCA (and all other layers on the CPU), we profile whether the input provides the same output as running the layer in CPU. Using the same inputs, we build a simple DNN classifier model (with three hidden layers), called *layer-classifier*, to output this matching result. The final output is denoted by $\{m_1^i, m_2^i, ..., m_j^i, ..., m_L^i\}$ $(m_j^i = \{0, 1\})$.

Also, we profile that for a given input, whether a model's outputs are the same if it runs in CPU and LCA entirely. The profiling is conducted for a list of inputs for each model running on a specific device. Using the same inputs, we build another simple DNN classifier model, called *model-classifier*, to output this matching result.

4.3.3 Heuristic Methods

Heuristic for Individual Layer Assignment

The design of the heuristic is based on Observation 2. For a given input to a DNN model, FLEX's *layer-classifier* can determine if the outputs of the model will match when each layer runs on the CPU and the LCA. Among the layers that have layerwise-matched results, FLEX moves the layers one by one to the LCA until the accuracy and timeliness requirements are met, or the LCA memory is full. If the requirements are still not met after all of these layers are moved to LCA, FLEX will move other unmatched layers to LCA. FLEX repeats the movements until the accuracy and time requirements are met, or the LCA memory is full.

When moving layerwise-matched layers or unmatched layers from CPU to LCA, FLEX first picks up the layer that generates a higher reduction in inference time and a lower reduction in accuracy to move out. For each layer l_k, after we move the layer from CPU to LCA, a new layer assignment $L_{l_k}^i$ is generated. FLEX first estimates the inference time, $T(L_{l_k}^i)$, and accuracy, $A(L_{l_k}^i)$, using *RF-T* and *RF-A*, respectively. Then, it

calculates the objective metric using $O^i_{l_k}$ using Equation (4.2). Finally, FLEX chooses the l_k among the layers, which results in the highest $O^i_{l_k}$ value.

This approach still suffers from a few shortcomings. First, even after allocating all possible layers to the LCA, the latency guarantees may not be satisfied. This is mainly due to the fact that the layer-based heuristic is only able to determine whether the outputs match or not along with confidence. To attain the latency guarantee, it is necessary to move layers to the LCA even if the outputs don't match, thereby sacrificing a bit of accuracy. Second, making a decision at the layer granularity means that the layer assignment may result in multiple communications between the CPU and the LCA, which can be prohibitively expensive. Thus, a layer-wise approach alone may not be efficient for the problem at hand. In order to account for the shortcomings, we propose a new model-splitting technique, which we describe below.

Heuristic to Find a Model Division Point

This proposed heuristic is mainly based on Observations 3 and 4. It combines the individual layer assignment heuristic and the key observations we make in sec:motivation to assign a consecutive block of layers to the LCA. Specifically, we bias the split towards maximizing accuracy or minimizing latency and use the individual layer assignment heuristic to determine the split.

FLEX considers whether the user prioritizes accuracy (i.e., maximizing accuracy while satisfying the time requirement) or time (i.e., minimizing time while satisfying the accuracy requirement) for a model to meet the requirements. If the user prioritizes accuracy, since shallow layers lose less accuracy, FLEX moves shallow layers from the first layer to LCA until the time requirement is met or the LCA memory is full. To expedite the process, FLEX moves M layers group by group, and when the time requirement is met, in the last group, FLEX moves layers from the end one by one to find the division point that meets the time requirement.

If the user prioritizes time for a model, FLEX moves layers from the end since heavy layers reduce more time. Specifically, it judges whether the model results on the CPU and LCA match each other using the *model-classifier*. If yes, FLEX moves the heavy layers to LCA as much as possible (while still satisfying the memory constraint) since it is unlikely to affect the result. Otherwise, FLEX moves the heavy layers as long as the accuracy requirement is satisfied and the memory constraint is satisfied until it reaches the division point layer κ, which saves the most time. Specifically, because the heavy layers are at the end, FLEX moves the κ starting from the $L/2$ until the end one layer by one layer and calculates T_s and checks the accuracy and time each time using *RF-A* and *RF-T*. Finally, FLEX finds out this point that meets the inference accuracy

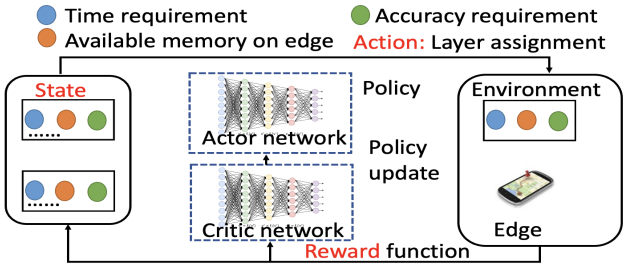


Figure 4.13: RL for determining the DNN layer assignment.

requirement and minimizes the time. However, this linear search is not time-efficient, so we propose a method to expedite the search as follows.

Expedite Searching the Model Division Point. FLEX expedites searching the model division point by leveraging the general trend of increasing saved time and accuracy as the layer index of the division point increases (Observation 4). We could use the binary search method to find such a division point. However, binary search requires that the numbers be ordered, which is not true in our case. We found that there is a general increasing trend, but a few layers are exceptional (such as the last four layers in Figure 4.4b). To solve this problem, we group a range of consecutive layers and calculate each group's average saved time and accuracy so the average values will be ordered. We then use the binary search to find the group with the highest saved time while satisfying the accuracy requirement. Next, we use the linear search in the identified group to find the optimal division point.

4.3.4 RL-based Method

The decision-making time overhead of the heuristic would be high (0.231 seconds shown in sec:exp). To handle this high overhead, we propose an RL-based method (as shown in Figure 4.13). FLEX builds one RL model for each DNN model structure (e.g., one RL model for VGG-16 and VGG-19). The RL model chooses the layer assignment for an input. FLEX further uses curriculum learning [7] for the generalization of the RL-based approach for different models as well as faster convergence. The RL-based method reduces the decision-making time of the heuristic by 90% (sec:exp). In the following, we first introduce the state, action and reward components of the RL model.

State space. The state space ($s \in S$) consists of the input's time and accuracy requirements, and the available memory of the device.

Action space. The action space is represented by $a \in A$, where an action, a, represents a layer assignment.

Reward. The reward function is given by:

$$r(s,a) = \begin{cases} -\gamma, & \text{if violates a constraint,} \\ \rho \times \sum_{i=1}^{n} \frac{O^i}{n}, & \text{otherwise} \end{cases}$$

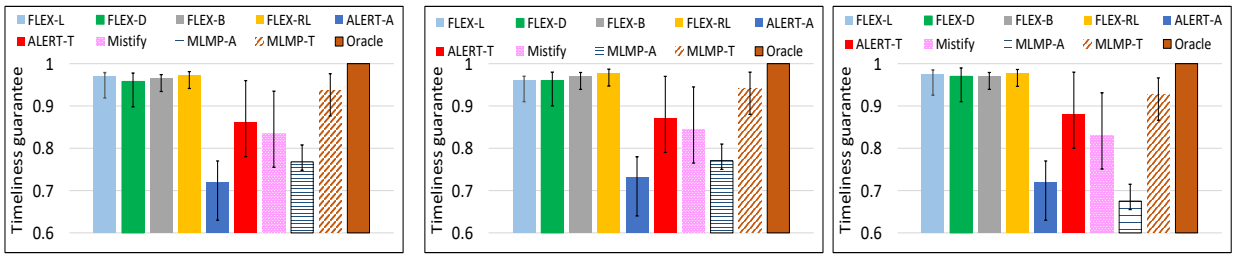
where ρ is a coefficient to control the reward and γ is a large negative constant reward to ensure that an allocation violating a constraint is not valid.

We use the Markov-chain soft actor-critic (SAC) [38] based RL. SAC is better than the other policy-based RL methods because it explores new states; thus, it avoids retraining for policy updates. In this framework, the actor aims to maximize expected reward while maximizing entropy, i.e., acting as randomly as possible in exploration.

In conventional training, training RL models for each DNN model and dataset individually can impose substantial overhead and result in RL models that are specific to each DNN model. To promote generalization across different models and expedite convergence, we employ curriculum learning [7]. Unlike the traditional RL approach, which randomly feeds training data, curriculum learning organizes and sequences the data following a specific distribution. This organized data feed leads to more effective and generalized model training. In our context, we observed that ordering the data based on the sequence of different DNN models, arranged by their complexity (e.g., from VGG-16 to VGG-19), yields the most substantial improvement in rewards. This sequential training approach allows RL to leverage the knowledge gained from training on less complex models to guide the training of more complex models. By systematically feeding data in this manner, RL training becomes more efficient, significantly enhancing the division point. To incorporate the complexity of the models into the curriculum learning process, we augment the state space with information about the number of layers, structural type (e.g., convolution, dense), and structural details (e.g., the number of filters and filter size for convolutional layers). Furthermore, to fully leverage curriculum learning, we train a single RL model for a group of models with similar structures (e.g., VGG-16 and VGG-19, ResNet-34 and ResNet-51). This grouping enhances the efficiency and effectiveness of the training.

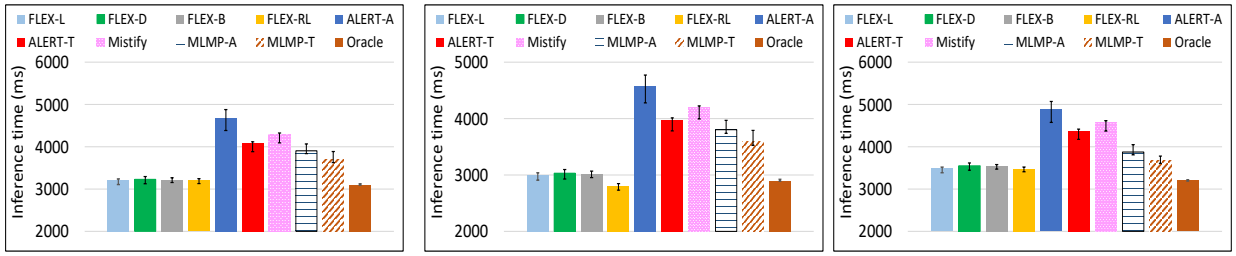
4.4 Implementation

We train the models in Table 4.1 using the Keras [18] and Scikit-learn library [86]. While these libraries provide APIs for constructing models, and numerous parameters such as learning rate and batch size, the models still require fine-tuning of the parameters to be well-trained. To achieve optimal parameter settings



(a) GPU 1. (b) GPU 2. (c) TPU.

Figure 4.14: Timeliness guarantee.



(a) GPU 1. (b) GPU2. (c) TPU.

Figure 4.15: Inference time.

(learning rate, batch size) to construct the well-trained models, we employed the grid search method [68], that systematically explores different combinations of parameters to identify the most effective configuration.

In order to profile the performance of the DNN models, we utilized images from the dataset specified in Table 4.1. These images required the DNN model to identify ten distinct types of objects, with each image typically containing just one kind of object. Given the possibility that the model might overlook an object in the image or classify an object erroneously, we adopted top-1% accuracy as the metric for performance evaluation. We employed the Android profiler app [17] to measure the usage of the LCAs using the metrics including the computation time and communication time, and memory utilization.

To run the divided DNN models on the CPU, we use the Tensorflow deep learning framework. Since our used smartphones use the Android system, we cross-compile the Tensorflow framework using NDK. To run DNN model layers on LCAs, we used the Android Native Development Kit (NDK) [4] to implement more flexible functions as a wrapper, which can run specific model layers on CPU or LCAs on the top of CUDA SDK [84].

4.5 Performance Evaluation

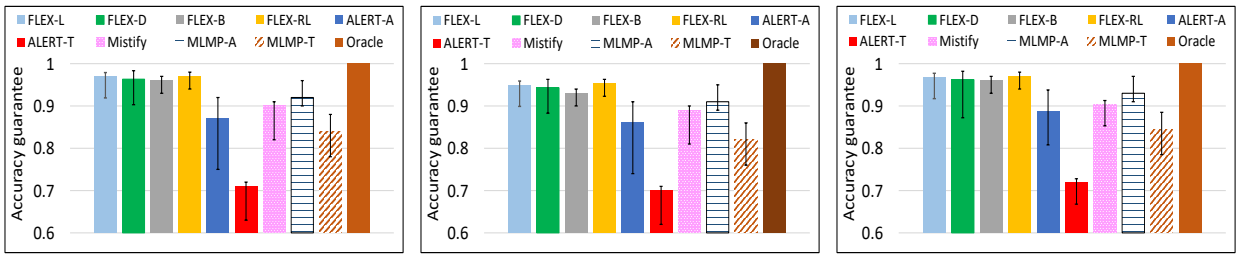
4.5.1 Experiment Settings

Our evaluation was conducted on three smartphones: GPU1, TPU and GPU2. The details of the GPU1 and the TPU are provided in sec:motivation. The GPU2 device is equipped with an 888G 4-core CPU with 2.8 GHz clock speed and an Adreno 680 4-core GPU along with 128GB storage and a Hexagon 780 AI engine that can support up to 32-bit floating point operation if configured. For these smartphones, the precision point for each DNN model was randomly selected from either 16-bit floating point or 8-bit integer precision settings. Unless otherwise mentioned, all the settings are the same as in sec:motivation. The number of inputs processed by each model was chosen randomly from the range [100,500]. The deadline for an input of one model was randomly chosen from $[0.5, 5]$ seconds and their accuracy requirement over 100 inputs was randomly chosen from $[80\%, 95\%]$, One input's accuracy is determined when the output matches to that of the true label for the experiment. We empirically set $\alpha = 0.5$ in the heuristic, $\gamma = 100$ and $\rho = 1$ in the RL-based method.

Comparison methods. We compared FLEX with MLMP-T and MLMP-A introduced in sec:motivation. As a reference, we also compare FLEX with *Oracle* and the following two model selection methods (which are not for layer assignment).

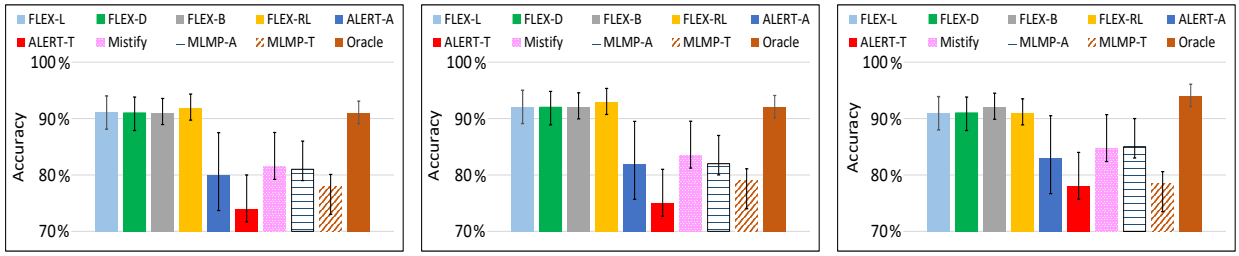
- *ALERT* [116] selects a model from prior selected model structures for a specific task to meet latency, accuracy, and energy constraints using an optimization problem. In our experiments, we let ALERT choose from all the models in Table 4.1 that can be for the task. Then, we randomly divide the model between CPU and LCA. We created two variants: ALERT-T and ALERT-A, as defined for MLMP.
- *Mistify* [37] chooses a model from a list of compressed models (with pre-known accuracies) according to the available resources of a device to meet the accuracy requirement. In our experiments, we created two compressed models for each model in Table 4.1 for Mistify to choose from and put layers in LCA as much possible until GPU memory is full.

Both of these methods select models from the Table 4.1 specifically limiting their choices to models within the corresponding task category. We compared these methods with the heuristic for individual layer assignment (FLEX-L), the heuristic for division point (FLEX-D), and its enhancement that uses binary search for the division point (FLEX-B) and RL-based layer assignment method (FLEX-RL), respectively.



(a) GPU 1. (b) GPU 2. (c) TPU.

Figure 4.16: Accuracy guarantee.



(a) GPU 1. (b) GPU 2. (c) TPU.

Figure 4.17: Accuracy.

4.5.2 Evaluation Results

Timeliness guarantee ratio and inference time. Figure 4.14 shows the timeliness guarantee ratio for all devices, which is calculated as the ratio of the number of inputs meeting their deadlines to the total number of inputs. For all the devices, the performance follows: *Oracle*>FLEX-L≈FLEX-RL>FLEX-B≈FLEX-D>MLMP-T>ALERT-T>Mistify>MLMP-A>ALERT-A.Figure 4.15 shows inference time for all the models. The result follows: *Oracle*<FLEX-L<FLEX-RL≈FLEX-B<F-LEX-D<MLMP-T<MLMP-A<ALERT-T<Mistify<ALERT-A.

FLEX-D has timeliness guarantee ratios 11-15%, 15-18%, and 25-33% higher than ALERT-T, Mistify, ALERT-A, and 10-13% and 26-35% higher than MLMP-T, and MLMP-A, respectively. It has an inference time 19-28%, 29-34%, and 23-32% lower than ALERT-T, Mistify, ALERT-A, and 11-18% and 13-21% higher than MLMP-T, and MLMP-A, respectively. FLEX-D outperforms Mistify and Alerts because FLEX-D is designed to determine layer assignment between the CPU and the LCA, while Mistify and ALERT are proposed to choose a model structure, they only randomly divide the model in our experiments. FLEX-D outperforms MLMP since MLMP is not input aware, so its optimal layer assignment for a model may not be optimal for a particular input. Also, it first finds a few assignment schedules to choose from, so it may miss a better assignment. Further, it may bias one metric at the significant cost of another metric by sorting layers based on one metric for movement between CPU and LCA.

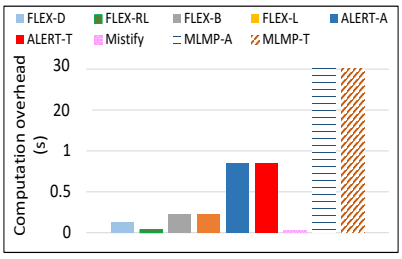


Figure 4.18: Computation overhead.

Figure 4.19: Accuracy per-layer prediction per-input.

Figure 4.20: Performance of model classifier.

FLEX-L and FLEX-RL have 2-3% higher timeliness guarantee ratio than FLEX-D, FLEX-L has 2-5% lower inference time than FLEX-D, while FLEX-RL generates similar inference time as FLEX-D. Compared to FLEX-D, which only determines a division point, FLEX-L's more fine-grained layerwise assignment method can help find a better layer assignment that generates a higher timeliness guarantee ratio or lower inference time. FLEX-RL automatically makes the decisions using the trained RL and sometimes could find better solutions than the heuristic FLEX-D. FLEX-B improves FLEX-D by 2% in timeliness guarantee ratio and by 3% in inference time. These results indicate the higher time efficiency of FLEX in expediting the search of the model division point. MLMP-A and ALERT-A perform the worst because they aim to improve accuracy instead of inference time. *Oracle* always meets the timeliness guarantee because it knows the best division decision and *Oracle* has 5% higher timeliness guarantee ratio and 6% lower inference time than FLEX-RL.

Accuracy guarantee ratio and accuracy. Accuracy is the percentage of the DNN outputs that match the true labels in the 100 inputs of a model. Accuracy guarantee is calculated as the ratio of the 100 inputs for a model whose accuracy requirements are satisfied. Figure 4.16 shows the accuracy guarantee ratio. The result follows: *Oracle*>FLEX-RL>FLEX-L≈FLEX-D≈FLEX-B>Mistify≥MLMP-A>ALERT-A>MLMP-T >ALERT-T. Figure 4.17 shows the accuracy for all the models for 100 requests. The order of the performance is as follows: *Oracle*>FLEX-RL>FLEX-L≈FLEX-D≈FLEX-B>MLMP-A>Mistify>ALERT-A>MLMP-T>ALERT-T. We see that FLEX-L, FLEX-D, and FLEX-B achieve similar performance. The result means that FLEX-D does not compromise accuracy, though it reduces the overhead of layerwise FLEX-L by only determining a division point. Also, our method for expediting searching the model division point also won't compromise the accuracy. In the following, we use FLEX-D to represent these three FLEX methods to compare with other methods. FLEX-D has 5-11%, 8-13%, 11-14%, 14-27%, 17-31% higher accuracy guarantee ratios than Mistify, MLMP-A, ALERT-A, MLMP-T, ALERT-T respectively, due to the same reasons explained in Figures 4.14 and 4.15. FLEX-RL performs 3-4% better than FLEX-D, which indicates that FLEX-RL can even outperform the heuristic method in accuracy. *Oracle* has approximately 4.5% higher accuracy guarantee ratio and 3% higher accuracy than the FLEX-RL.

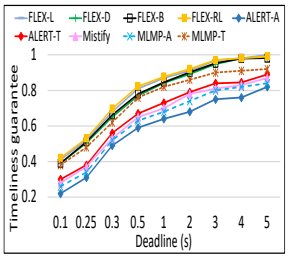


Figure 4.21: Timeliness scalability.

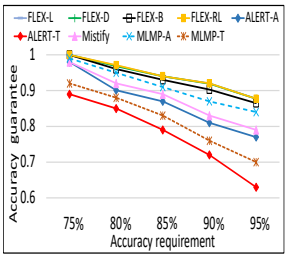


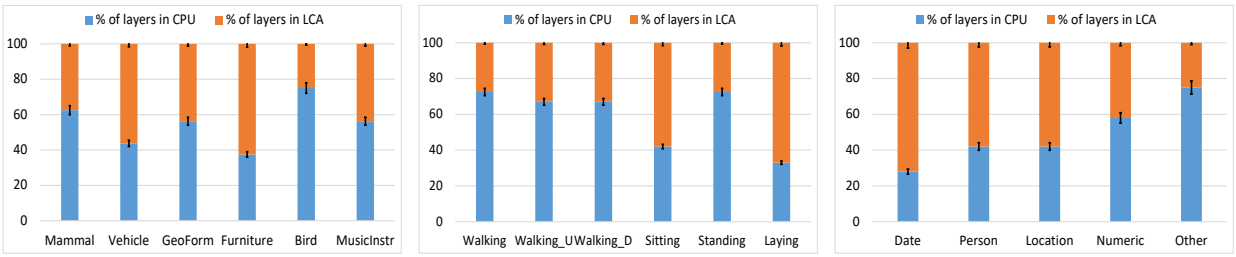
Figure 4.22: Accuracy scalability.

Computation overhead. Figure 4.18 shows the computation overhead of each method, which is the decision making time, i.e., the time duration from when an input arrives to when it starts the processing. The result follows: Mistify<Flex-RL<Flex-B<Flex-D<Flex-L<ALERT-A≈ALERT-T<MLMP-A≈MLMP-T. Mistify has the lowest computation overhead because it only chooses a compressed DNN model based on the known accuracies of the models. ALERT-A and ALERT-T need to solve an optimization problem for model selection. MLMP-T and MLMP-A take the longest decision-making time because they generate a list of possible combinations and then search for the best combination from the list. Flex-D has 48% lower computation overhead compared to Flex- since it finds one division point instead of individual layer assignment. Flex-B improves Flex-D by 41% since it runs binary search instead of linear search, and Flex-RL improves Flex-B by 90% since it uses RL.

Energy consumption. We further measure the energy consumed on the GPU smartphone by monitoring its battery using an app named AccuBattery [26]. Table 4.2 shows the total energy consumed for decision-making and inference for all the inputs. The order of the consumed energy follows the same order as the inference time since more processing time on an input consumes more energy on the device.

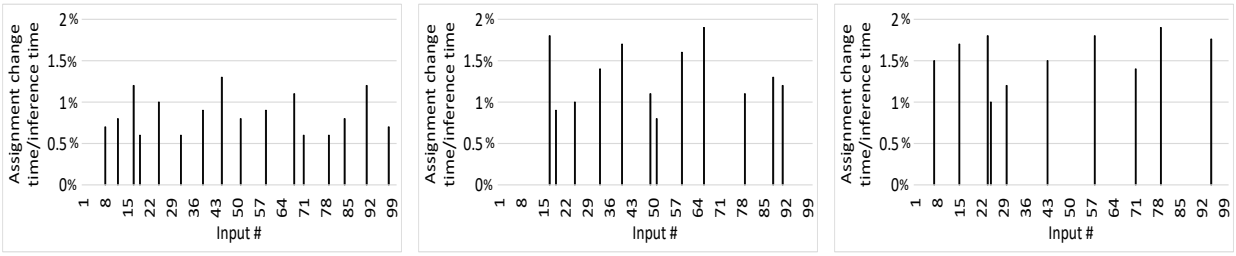
Performance of estimators. To build the *RF-T*, *RF-A*, *layer-classifier*, and *model-classifier* models, we used 70% of profiled data for training, and the rest of the data for testing. The training took approximately 5-6 hours for *RF-T* and *RF-A*, 2.45 and 4.2 hours for *layer-classifier* and *model-classifier*, respectively.

In our experiments, *RF-T* and *RF-A* show 7.14% and 9.56% Mean Average Percentage Error (MAPE) values in terms of predicting the inference time and accuracy, respectively. The error contributes to the lower performance of Flex compared to *Oracle*. Figure 4.19 shows the accuracy of the per-layer CPU/LCA result matching prediction by the *layer-classifier* of Flex for the VGG-16 model. The accuracy varies between 87-93% for all the layers. Figure 4.20 shows the accuracy of the per-model CPU/LCA result matching prediction by the *model-classifier* of Flex for the models in Table 4.1. The accuracy varies between 84-92% for all the models.



(a) ImageNet. (b) HAR. (c) SQuAD.

Figure 4.23: Input awareness of FLEX.

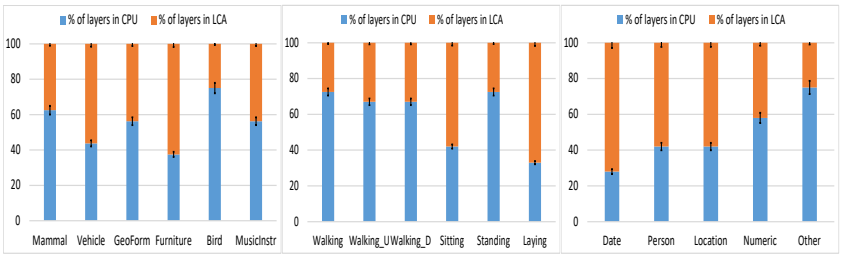


(a) ImageNet. (b) HAR. (c) SQuAD.

Figure 4.24: Layer assignment of FLEX.

Performance of RL training. The convergence of FLEX's RL training is faster than traditional RL training (where training data is fed randomly) because of the curriculum learning. We also used the traditional training method to train the RL model. For the two structures of the VGG model, the convergence speed of the proposed RL is approximately 7 hours compared to a total of 16 hours for traditional training on a machine with 4 Nvidia RTX 2080Ti GPUs.

Scalability testing. Figure 4.21 shows the timeliness guarantee of different time deadlines for a fixed accuracy requirement of 85%. The figure shows that the timeliness guarantee increases for FLEX 1.44× when the deadline increases from 0.1 to 5 seconds. The timeliness guarantee for FLEX is above 0.91 for any deadline above 2 seconds. All other methods also follow the same trend of increasing timeliness guarantee with the deadline. The order of performance for all the methods for timeliness guarantee remains the same as Figure 4.14. Figure 4.22 shows the accuracy guarantee for different accuracy requirements varying from 75% to 95% with a 5% increase in each step for a fixed deadline of 2 seconds. We observe that the accuracy guarantee decreases gradually as the accuracy requirement increases for all methods, and it decreases from 1 to 0.88 for FLEX when the accuracy requirement is increased from 75% to 95%. FLEX still achieves 0.94 accuracy guarantee ratio for the accuracy requirement of 90%. The order of performance for all the methods remains the same as in Figure 4.16.



(a) Imagenet. (b) HAR. (c) SQuAD.

Figure 4.25: Layer assignment of FLEX.

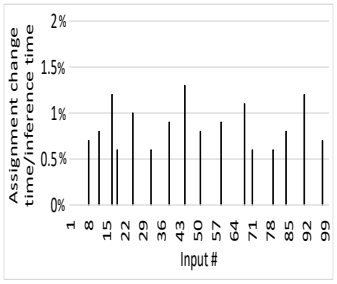


Figure 4.26: Layer assingment.

Table 4.2: Total consumed energy.

Method	Decision making (mW)	Inference (mW)	Total (mW)
FLEX-D	6	117	123
FLEX-RL	2	91	93
FLEX-B	5	116	121
FLEX-L	9	119	128
ALERT-A	11	212	223
ALERT-T	11	170	181
Mistify	1	192	193
MLMP-A	19	129	148
MLMP-T	19	124	143

Input-aware layer assignment. Figure 4.23 shows the average percentage of layers assigned to CPU and LCA for different data categories, respectively. for different datasets with an accuracy requirement of 85% and a deadline of 3 seconds. We selected inputs from diverse categories in the datasets. These figures illustrate that individual categories exhibit varying layer assignments. For instance, the "Mammal" category displays 43.75% more layers assigned to the CPU compared to the "Vehicle" category. This divergence in layer assignment percentages between the CPU and LCA exemplifies the input-awareness of FLEX.

Change of Assignment. We further measured the time that FLEX-RL used to load the layers each time for all the datasets and plotted it in Figure 4.26. For 100 inputs of ImageNet, HAR and SQuAD, the division point needed to be changed 17, 12, and 10 times, and the time for reassignment of the layers was 0.138%, 0.168% and 0.156% of the inference time, respectively.

4.6 Related Work

Model Selection and Compression for Inference. There is a group of methods that select [116, 93, 36, 121, 37, 124, 125] from a group of pre-selected models or generate [148] new model from unit blocks based on currently available resources of the device in order to meet requirements such as accuracy, deadline, and energy. Another group of methods [70, 141] chooses model compression techniques prior to executing a DNN model on edge devices to find an optimal balance between latency and energy cost for specific resource constraints.

Executing Inference on Edge. Long *et al.* [71] proposed MEANet that classifies the inputs into easy and hard classes and assigns them to the edge and cloud, respectively. Kang *et al.* [52] proposed a model parallel pipeline scheme, LaLaRAND, which tries to balance the execution time (computation and communication) between CPU and GPU, and quantizes CPU-based layers. Kim *et al.* [55] proposed a tensor-parallel quantization scheme, μLayer, that divides each layer of a DNN model between CPU and GPU and then performs quantization on the model to reduce the execution time. Xu *et al.* [126] proposed ApproxDet for object detection from video applications, which dynamically set parameter values for sampling interval and tracker type based on the runtime content and resource availability to optimize the accuracy and latency. Seo *et al.* [99] proposed a scoring scheme, which ensures that the inputs with low latency SLOs are prioritized to the fast processors (GPU vs. CPU) and have low wait time.

Some methods for the clusters [77, 142] focus on fully utilizing the CPU and GPU resources in order to reduce training or inference time but do not consider the low accuracy issue of LCAs.

In spite of many methods for inference jobs on edge devices, within our knowledge, MLMP [107] is the only method that schedules layers of a model across the CPU and LCA, keeping accuracy in mind. However, as discussed in sec:introduction and sec:motivation, MLMP has some shortcomings. FLEX overcomes the shortcomings and achieves significantly higher performance on accuracy, inference time, and energy consumption.

4.7 Limitations and Discussion

- **Offline training time.** The classifier and RL models are trained offline for a fixed set of the DNN models, which took 2.45 and 7 hours, respectively. The offline training won't affect the online inference time. The cost of training the regression and RL models on the AWS with m5-instance would be approximately 10 and 30 dollars, respectively. Considering the superior performance of FLEX over the state-of-the-art methods, the cost of training these models is worthwhile.
- **More accurate estimators.** Recall that our estimators for estimating accuracy, inference time and CPU/LCA result matching have a 6% gap with *Oracle* We will seek more accurate models for the estimation.
- **Pipelining.** We will pipeline the inputs across CPU and GPU to reduce the inference time of a batch [113]. While an inference model is running on consecutive inputs, the inputs can be pipelined [113] across CPU and GPU to improve the inference time. While the computation of i^{th} input is running on the GPU after finishing on the CPU, the computation of $(i+1)^{th}$ input can run on the CPU simultaneously.

- **RL vs. heuristic.** Although FLEX-RL achieves better performance than the heuristics of FLEX, if a model with a different structure arrives, FLEX-RL may not find the optimal layer assignment. In this case, FLEX-B can be used, and the collected data will be used for training the RL for the future use of FLEX-RL for this model.
- **Very tight deadline.** As other state-of-the-art works [126, 116, 37], FLEX cannot support very tight deadlines (e.g., 0.01s). We will work on reducing the decision-making time and layer assignment change time to further reduce inference time.
- **Dynamic precision point selection.** We will further improve FLEX to dynamically decide the precision point for each layer. AMPT [75] adopts a precision point setting strategy during training, while we will focus on inference.

4.8 Conclusion

A large portion of smartphone users in the world own low-cost edgedevices, and modern accelerators often use lower precision point arithmetic in order to save energy, thus becoming LCAs. The LCAs sacrifice accuracy although they can substantially accelerate ML tasks. In this paper, we present FLEX, which dynamically executes a model between a device's CPU and LCA to achieve both high accuracy and lower latency on LCAs. FLEX utilizes a DNN model to predict whether executing a layer with the given input will result in the same result in the CPU and LCA and then decides whether it runs on the CPU or LCA. To reduce the overhead for the layer-wise assignment, FLEX has an alternative solution that decides one division point of the DNN model between the CPU and LCA. Moreover, FLEX has an RL-based approach to further reduce the decision-making time and switches to the RL approach from the heuristic when the RL model is well-trained. Our experiments on real devices show that FLEX achieves superior performance than state-of-the-art approaches.

Chapter 5

Mitigating KV Cache Competition to Enhanced User Experience in LLM Inference

The chapter handles the challenge highlighted in Chapter 1: LLM KVC management for Edge Devices. This chapter integrates several strategies for effective KVC allocation and reduce latency during LLM inference in case of demand based KVC allocation. Our proposed system, CacheOPT targets to allocate an equal amount of KVC based on the prediction of the response length. Then, it dynamically adjusts the amount of allocated KVC on top of the predicted output length through confidence score and then decide the padding based on arrival rate to better utilize the GPU memory resource, by reducing memory violation severity regarding the allocated KVC. CacheOPT also prioritizes request eviction or preemption based on resource occupancy and projected completion time, ensuring efficient resource distribution. Additionally, CacheOPT employs a cost-based heuristic to choose between either recomputation or swapping to reduce latency, while considering the system loads and the request characteristics.

5.1 Introduction

The Large Language Model (LLM) models have found widespread applications across various domains, including automated customer support, chatting, and real-time translation. However, deploying LLMs in real-world applications often comes with high tail Time-to-First-Token (TTFT) or Time-Between-Tokens (TBT) [95, 106, 151], which can impair user experience, especially in time-sensitive tasks. Reducing TTFT

and TBT is crucial for ensuring seamless user interactions and satisfied user experience. It also enables the broader adoption of LLMs in different applications.

Due to limited GPU memory and the large volume of KV-cache (KVC) values, KVC becomes a bottleneck, significantly impacting both TTFT and TBT. The first iteration-level scheduler Orca [135] pre-allocates KVC for the maximum sequence length for each request. However, this approach wastes memory [49], limits the batch size and hence GPU utilization (e.g., 0.4% [49]) and throughput. To address this problem, vLLM [60] uses block-based KVC allocation approach, in which, a block, consisting of a fixed number of tokens, is allocated to a request each time. To form a batch in each iteration, the requests from the waiting queue are added to the batch until the whole KVC is allocated. While this approach significantly reduces KVC reserved waste, a running request may experience block allocation failure when it exhausts its allocation. In this case, the last arrived running request is selected to preempt based on the first-in-first-serve (FCFS) policy. Both approaches introduce delays, increasing TBT.

Another method allocates KVC to a request equal to the response length predicted by an LLM model [146, 50]. Compared to the block-based allocation approach, it reduces the chances of KVC allocation failures but still leads to certain reserved waste since allocated KVC is not used instantly and generates KVC allocation failures with underestimation. [146] proposes to add a constant padding (e.g., 100 tokens) to the predicted value to avoid underprovisioning, indicates that overprovision is not as important as underprovision.

Different LLM applications and users often have varying SLOs for TTFT and TBT. For example, real-time translation applications prioritize ultra-low TTFT to deliver the first token as quickly as possible, enabling smoother conversation flow. In contrast, document summarization may tolerate higher TTFT in exchange for lower TBT. Moreover, users' reading speeds influence their tolerance for TBT. A fast reader using a summarization tool may prefer shorter TBT to maintain a seamless reading experience, while a slower reader may be less sensitive to longer TBT but require low TTFT for responsiveness. The diverse demands necessitate adaptable LLM systems capable of meeting varying TTFT and TBT SLOs.

In this paper, we aim to mitigate KV cache competition while enhancing user experience by satisfying the diverse TTFT and TBT SLOs through studying three problems:

(1) How to determine the padding size to ensure that the KVC demand is satisfied with a high specified probability?
(2) How to allocate KVC to a request initially and during token generation process?
(3) How to choose requests to preempt?

(4) How to choose a preemption strategy between swapping and recomputation to reduce the preemption time (defined as the time that a request is halted)?

To achieve this goal, we conducted an experiments based on real traces and made the following observations (Os):

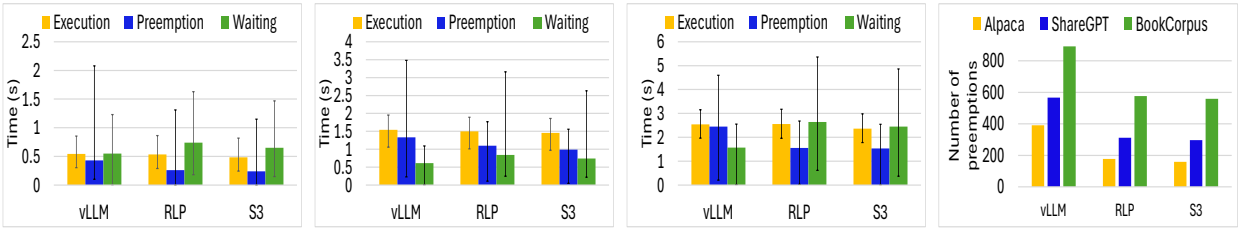
(1) Block-based KVC allocation increases preemptions and TBT, while prediction-based allocation raises TTFT. A new method is needed to balance both.
(2) Previous prediction-based methods often cause underprovisioning (increasing TBT) and overprovisioning (increasing TTFT). The padding size must be carefully determined, considering the request arrival rate, to balance TTFT and TBT.
(3) FCFS-based preemption results in more preemptions than approaches that account for remaining time and KVC usage.
(4) For sequences exceeding the sweet spot length, swapping results in shorter times compared to recomputation.

Leveraging these observations, we propose a system, that optimizes the cache operations for mitigating KV cache competition (CacheOPT). We name this CacheOPT to reflect the system's focus on efficient cache usage. CacheOPT consists of the following components.

(1) **Confidence-based Padding.** We modify an LLM model to predict output length and deviation direction for adding or subtracting padding. Using Hoeffding's inequality theory [43, 8], we determine the padding needed to ensure a request's KVC demand is satisfied with a high specified probability and adjust it based on arrival rate to balance TTFT and TBT. (O2)
(2) **SLO-aware Batching and KVC Allocation.** We reuse allocated but unused KVC and select waiting and returned requests that must run to satisfy their TTFT and TBT SLOs. The remaining unallocated KVC is distributed among the selected requests to maximize throughput. Additionally, it proactively allocates KVC before instead of at the time a request exhausts its allocation and reserves KVC globally to prevent preemptions. (O1)
(3) **Preemption Policy.** We order requests for preemption based on their latency SLO, remaining completion time and KVC occupancy in order to avoid SLO violation and preemptions, and reduce preemption time. (O3)
(4) **Preemption Strategy Selection.** When the KVC size exceeds the observed sweet spot sequence length, we opt for swapping; otherwise, we choose recomputation. (O4)

Table 5.1: Trace properties and experiment settings.

Trace	Input length			Output length			Number of requests	Mean arrival rate
	avg	min	max	avg	min	max		
Alpaca	19.31	9	2.47K	58.41	13	292	52K	32
ShareGPT	161.31	16	3.2K	337.99	19	991	90K	28
BookCorpus	1952.11	18	461K	681.2	32	1041	11K	1.2



(a) Alpaca. (b) ShareGPT. (c) BookCorpus. (d) Number of preemptions.

Figure 5.1: Measurements for different traces for OPT-13B.

Experimental results show that CacheOPT achieves up to a 3.29× and 2.83× lower tail TBT and TTFT and 47% and 53% higher TTFT and TBT SLO attainments than the state-of-the-art methods, respectively. We will distribute CacheOPT's source code after the paper is accepted.

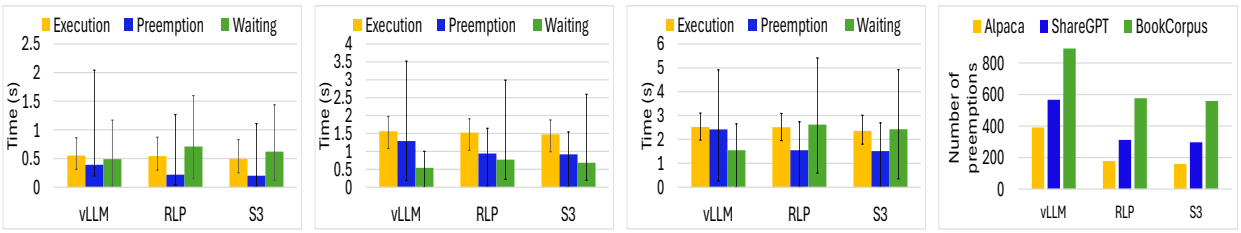
5.2 Experimental Analysis

5.2.1 Experiment Settings

Machine settings. We used an AWS p4d.24xlarge instance, which features 8 NVIDIA A100 GPUs. Each GPU has 80GB of memory. The GPUs are interconnected via a 600 GB/s NVSwitch. We executed the OPT-13B model [138] on a single GPU, and the OPT-175B model, partitioning the model across 8 GPUs with both model and tensor parallelism degree of 2 as in [135].

Request settings. We used the Alpaca [109], ShareGPT [100], and BookCorpus [56] traces. Table 5.1 shows their features and the mean request arrival rate setting. It follows a Poisson distribution [135, 60]. The block size was set to 32. For BookCorpus [56], we divided the prompts into 2048 tokens to meet the requirements of the LLM models. We used 3-hour trace for OPT-13B and 1-hour trace for OPT-175B.

Schedulers. We conducted the experiment measurements for the following methods. 1) Response Length Prediction (RLP) [146] uses the LLM to predict the response length from a request and always adds a padding of 100 tokens. It tries to schedule the requests with similar lengths in the same batch. Any underprovisioned requests are preempted and later executed when there is sufficient KVC. 2) S^3 [50] uses the LLM model to predict the response length bucket (with a 50-token increment) for each request and allocates the upper bound of the predicted bucket.If a request faces insufficiently allocated KVC, it is preempted, and its KVC demand



(a) Alpaca. (b) ShareGPT. (c) BookCorpus. (d) Number of prremptions.

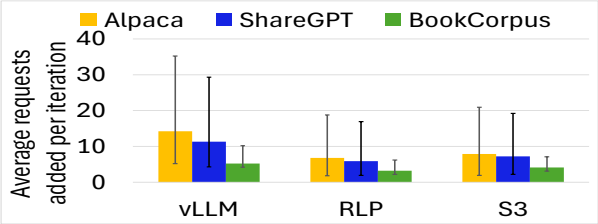
Figure 5.2: Measurements for different traces for OPT-175B.

is doubled for the next allocation. 3) vLLM [60] uses the block-based KVC allocation and FCFS-based preemption, and defers new requests until preempted ones are completed.

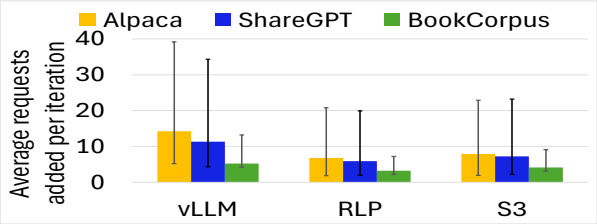
To make these methods comparable, we used our own fine-tuned OPT-13B model for the response length prediction in RLP and S^3. To avoid the interference on the LLM inference, we ran this model in another server. In this paper, *predicted output length* refers to the output length from the LLM, and *estimated output length* refers to the predicted output length adjusted with the padding.

5.2.2 Impact of KVC Allocation Methods on TTFT and TBT

Although [146] indicates that overprovisioning is less critical than underprovisioning, this claim primarily applies to requests with overprovisioning. However, overprovisioning reduces the number of requests accommodated in a batch, increasing waiting time and TTFT. A request's *waiting time* is the duration its prompt remains in the queue before execution begins; and its *execution time* is the duration from when it is dispatched to the execution engine until completion, excluding preemption time. Figures 5.1a-5.1c and Figure 5.2a-5.2c show the average execution, preemption, and waiting times for OPT-13B and OPT-175B, with error bars representing the min and max values. RLP and S^3 generate 29%-49% and 26%-45 lower preemption time but 43%-72% and 27%-65% higher waiting time than vLLM. This is because that in vLLM, each request is only allocated with a block at a time, and it has to be preempted upon KVC allocation failure when it uses up its allocated block. RLP and S^3 allocate to a request the KVC equal to its estimated sequence length, so they reduce the number of requests accommodated in a batch and increase the TTFT but reduce the premption time, which occurs only at underprovisioning. These findings are verified in Figure 5.2d, which shows the average number of preemptions, and Figure 5.3, which shows the average number of requests added to the batch per iteration, with error bars representing the minimum and maximum values. S^3 has 7% lower preemption time, 5% lower waiting time, and adds 12-17% more requests to the batch per iteration than RLP since S^3 adds 100-token padding while RLP's bucket size is 50. We calculated that the preemption time and the waiting time can take up to 71% and 75% of the execution time on average across the three datasets.

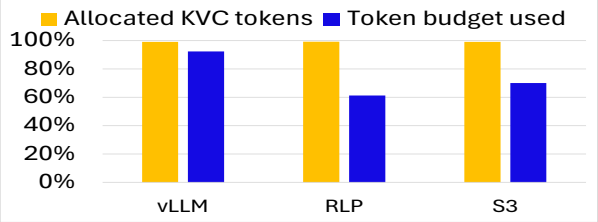


(a) OPT-13B.

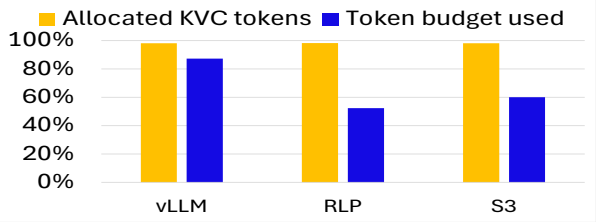


(b) OPT-175B.

Figure 5.3: Average requests added per iteration.



(a) OPT-13B.



(b) OPT-175B.

Figure 5.4: Illustration of resource bottleneck.

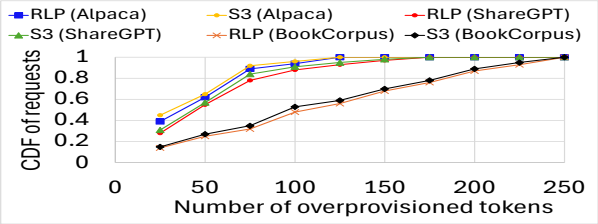
We find the *token budget* for the number of tokens in a batch to maximize throughput as in [1]. Figure 5.4 shows the average token budget used and the average number of allocated KVC tokens at each iteration. We see that while all systems fully alloctae the KVC, vLLM more fully utilizes the token budget than RLP and S^3 by 37% and 33%.

Theorem 5.1. *Due to KVC bottleneck, the block-based KVC allocation method increases preemptions and TBT, while the prediction-based KVC allocation method increases request waiting time and TTFT. A novel approach is required to limit both TTFT and TBT effectively.*

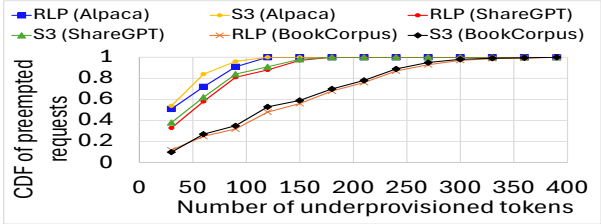
5.2.3 Padding Size Determination and Impact

Figures 5.5 shows the CDF of requests versus the number of over and under provisioned tokens for RLP and S^3. Overprovisioning and underprovisioning are consistently observed. Different requests exhibit varying degrees of these deviations, with longer prompts generally experiencing greater overprovisioning and underprovisioning amounts.

Figure 5.6 shows the response latency decomposed to wating time, preemption time and execution time versus different padding size at different arrival ratefor RLP. At each arrival rate, we observe that as the padding size increases, the preemption time decreases, the waiting time increases, and the execution time remains stable. The total response latency exhibits a concave pattern, initially decreasing with smaller padding sizes and then increasing beyond a certain threshold. Also, the arrival rate increase leads to higher waiting time. The padding size that minimizes the response latency varies for different arrival rates.

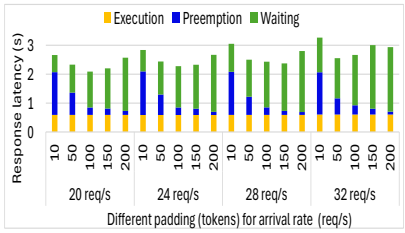


(a) Overprovisioning.

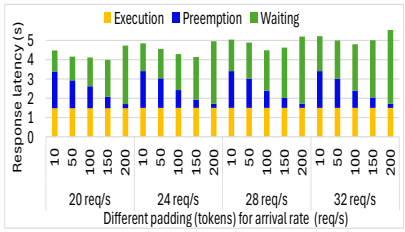


(b) Underprovisioning.

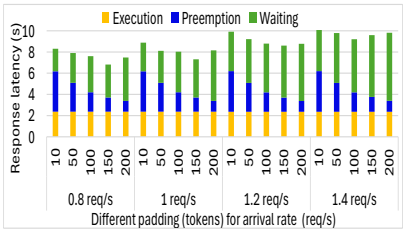
Figure 5.5: CDF of requests vs. over/under provisioned tokens.



(a) Alpaca.



(b) ShareGPT.



(c) BookCorpus.

Figure 5.6: Trade-off between preemption time and waiting time versus padding size for OPT-13B.

Theorem 5.2. *Previous prediction-based methods often cause underprovisioning (increasing TBT) and overprovisioning (increasing TTFT). The padding size must be carefully determined, conidering the request arrival rate, to balance TTFT and TBT.*

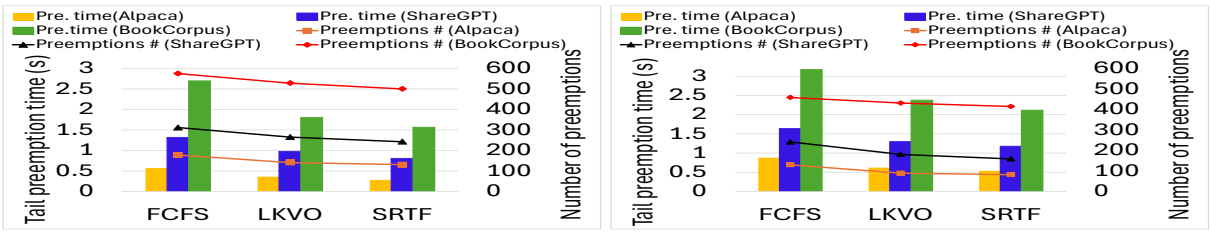
5.2.4 Preemption Policy

In RLP, we used the following preemption policies that preempt the request: 1) with the lastest arrival time based on FCFS employed in vLLM, 2) with the longest remaining time based on the Shortest-Remaining-Time-First (SRTF), and 3) with the smallest occupied KVC based the Lowest-KVC-occupancy (LKVO). Figure 5.7 shows the tail preemption time and total number of preemptions, respectively. FCFS exhibits 26%-58% and 39%-1.03× higher tail preemption time, 9%-31% and 13%-39% more preemptions compared to LKVO and SRTF, respectively. SRTF helps reduce KVC competition by making running requests release their KVC sooner, thus lowering preemptions. LKVO minimizes the time spent on swapping or recomputation.

Theorem 5.3. *FCFS leads to more frequent preemptions and increased preemption time compared to SRTF and LKVO.*

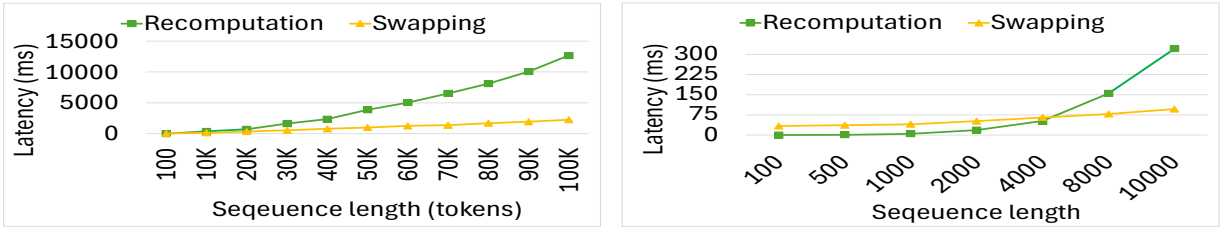
5.2.5 Preemption Strategy: Swapping or Recomputation?

The time complexity of swapping is $O(s)$ and that of recomputation is $O(s^2)$, where s is the sequence length. Figure 5.8 shows the latency for swapping including swappingin and out and recomputation for the varying sequence length. It confirms that swapping follows a linear growth, while recomputation follows a quadratic growth based on the sequence length. We observe that swapping is faster when the sequence length exceeds



(a) OPT-13B. (b) OPT-175B.

Figure 5.7: Performance of different preemption policies



(a) 100-100k sequence length. (b) 100-10k sequence length.

Figure 5.8: Latency of swapping and recomputation.

4000 tokens; otherwise, recomputation is faster. However, vLLM and RLP use recomputation and S^3 uses swapping as the default preemption strategy irrespective of the sequence length, and users can choose a strategy at a time.

Theorem 5.4. *Relying on a single preemption strategy is inefficient. When the sequence length exceeds a threshold (e.g., 4k in our setting), swapping is faster than recomputation.*

5.3 System Design

5.3.1 Solution Overview

Based on our observations, we propose CacheOPT. CacheOPT consists of the following components as shown in Figure 5.9.

(1) *Confidence-based padding* guided by O5.2 (5.3.2).

(2) *SLO-aware batching and KVC allocation* per O5.1 (5.3.3).

(3) *Preemption policy* guided by O5.3 (5.3.4).

(4) *Preemption strategy selection* guided by O5.4 (5.3.5).

In Figure 5.9, users' requests are initially entered to the waiting queue. *Confidence-based padding* (①) is executed to estimate the output length of each request when it waits in the queue. After each iteration, *SLO-aware batching and KVC allocation* (②) selects waiting requests to form a batch and allocates KVC to each batched request. Next, the batch is forwarded to the execution engine to be executed. When a request

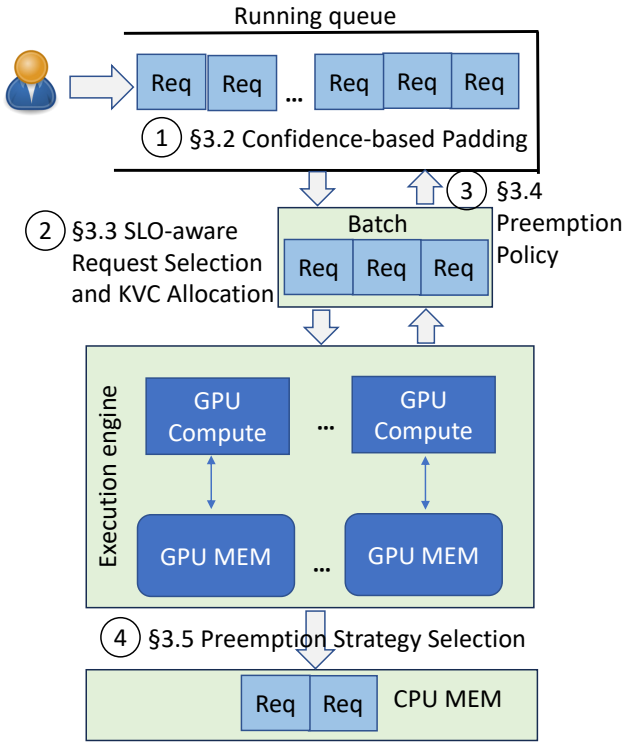


Figure 5.9: Architecture of CacheOPT.

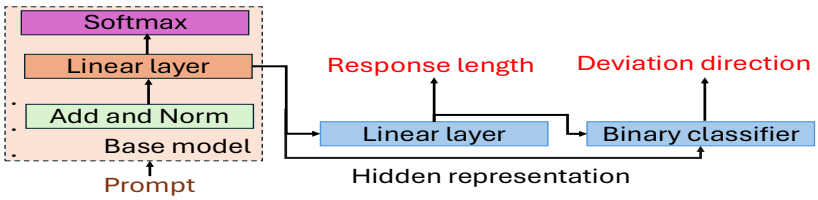


Figure 5.10: Our model for output length prediction.

experiences under-provisioning, the *preemption policy* (③) is executed to choose the requests to be preempted. The preempted requests are entered to the waiting queue and ordered. For a request to be preempted, the *preemption strategy selection* (④) is executed to reduce the preemption time of this request and other running requests.

5.3.2 Confidence-based Padding

Given the autoregressive nature of LLMs, accurately predicting the response length of a request poses a challenge. RLP achieves an accuracy of 67% [146], while S^3 reaches 79% [50], resulting in overprovisioning and underprovisioning and increasing TTFT and TBT (O5.2). To address this issue, we propose a confidence-based padding method, leveraging Hoeffding's inequality to bound overprovisioning and underprovisioning with high probability or confidence.

Output length prediction. Previous methods relying on LLMs to predict output length lack the ability to provide high confidence to bound the overprediction or underprediction, or to identify the deviation direction. Our method addresses these issues. As in [146], we use OPT-13B as the base model and extend its architecture to predict response length, and deviation direction. As shown in Figure 5.10, we add two layers (in blue) after the last linear layer in the base model. This layer outputs the hidden representation of the input prompt, which is the input of our added linear layer. The added binary classifier outputs the deviation direction (0/1).

The added linear layer predicts the output length. The binary classifier is a fully connected layer. Its input includes the hidden representation of the last linear layer of the base model and that of our added linear layer.

The input to the model is the prompt itself. The fine-tuning is conducted in two steps. First, we fine-tune the model without the binary classifier using the ground-truth of the output lengths. Then, using the fine-turned predictor, we predict the output lengths and collect the data of deviation direction. Second, we use this collected data to find-tune the whole model including the binary classifier.

Padding determination. For a given predicted output length, if its actual output lengths follow a specific probability distribution, the padding for the predicted output length can be determined based on this distribution to achieve a certain confidence. However, the probability distributions of the 9k fine-tuning requests and 6k inference requests does not follow a certain distribution. Therefore, this approach is not viable.

We then leverage the Hoeffding's inequality theory. It is used to bound the deviation with a specified confidence. Let $X_1, X_2, ..., X_n$ be independent random variables, where $a_i \leq X_i \leq b_i$ for $i = 1, 2, ..., n$. Let us define $S = X_1 + X_2 + ... + X_n$ and use $E[S]$ to denote the expected value of the sum. Then, for any limit $t > 0$:

$$P(S - E[S] \geq t) \leq \exp\left(\frac{-2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right) \tag{5.1}$$

We use X_i as the actual output length of request i, $\hat{X}_i$ as its predicted length, t_i is the maximum allowed deviation, and $(b_i - a_i)$ defines the range of request length for request i in the predicted values. Then, Equation (5.1) becomes:

$$P((X_i - \hat{X}_i) \geq t_i) \leq \exp\left(\frac{-2t_i^2}{(b_i - a_i)^2}\right) \tag{5.2}$$

Let c_i be the specified confidence, i.e., $P((X_i - \hat{X}_i) \geq t_i = 1 - c_i$. By solving the equation, we get:

$$t_i = \sqrt{-\frac{(b_i - a_i)^2}{2} \cdot \ln(1 - c_i)} \tag{5.3}$$

We use this t_i as the padding for underprediction. From Equation (5.3), higher c_i leads to more padding and vice versa. Padding t_i for overprediction is calculated similarly. Based on the deviation direction, the padding is added or subtracted from the predicted output length.

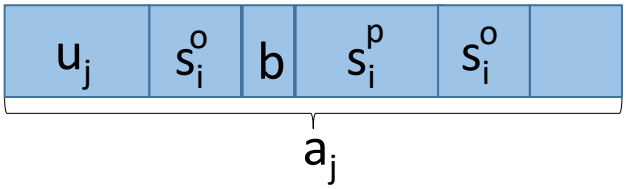


Figure 5.11: Emdedding method to reuse allocated but unused KVC.

Based on O5.2, a higher request arrival rate results in longer waiting times, necessitating a lower padding size hence lower confidence, and vice versa. Thus, we propose dynamically adjusting c_i based on the request arrival rate λ:

$$c_i = \frac{\alpha}{1 + \beta\lambda}, \tag{5.4}$$

where $0 < \alpha \leq 1$ controls the maximum possible confidence, and $\beta > 0$ regulates the sensitivity of the confidence to the arrival rate changes. At low arrival rates ($\lambda \to 0$), the denominator $1 + \beta\lambda$ approaches 1, resulting in $c_i \approx \alpha$. As the arrival rate λ increases, the denominator grows, causing c_i to decrease. This reflects reduced confidence in predictions when KVC is limited for the requests. The parameters here are empirically determined.

5.3.3 SLO-aware Batching and KVC Allocation

This component tackles the challenge outlined in O5.1 by incorporating (1) an embedding method to reuse allocated but unused KVC (5.3.3), (2) a request selection and KVC allocation strategy (5.3.3), and (3) preemption-avoidance mechanisms (5.3.3). Key notations are summarized in Table 5.2.

Table 5.2: Summary of notations.

Notation	Description
SLO_{ttft}	TTFT SLO.
SLO_{tbt}	TBT SLO.
$\mathbf{N}_w$	Waiting requests that must execute next to meet SLO_{ttft}
$\mathbf{N}_r$	Returned requests that used up allocated KVC but must execute next to meet SLO_{tbt}
$\mathbf{N}'_w$	Selected waiting requests that contribute to exhausting the token budget
$\mathbf{N}'_r$	Returned requests that used up allocated KVC but not needing immediate execution
T_I^{max}	Maximum iteration latency historically observed
s_i^p	Prompt length of request i
s_i^o	Estimated output length of request i
M_i	Total KVC demand for request i: $M_i = s_i^p + s_i^o$
A'_{kvc}	Unallocated KVC available for the current iteration
B	Fixed block size for KVC allocation
D_{kvc}	Total KVC demanded for critical requests: $D_{kvc} = \sum_i^{\lvert\mathbf{N}_w\rvert}(s_i^p + B) + \sum_i \lvert\mathbf{N}_r\rvert B$
a_i	Allocated KVC for a running request i
u_i	Currently used/occupied KVC for a running request i
γ	Factor used to determine KVC allocation cuts based on SLO priority
E_{kvc}	Excess demanded KVC: $E_{kvc} = \sum M_i - A'_{kvc}$

Embedding Method

As shown in Figure 5.11, a running request r_j's allocated KVC is denoted by a_j and its currently used KVC is denoted by u_j. If another request r_i uses r_j's allocated but unused KVC starting from the location marked by the red line, when both r_i and r_j run s_i^o iterations, r_i completes and releases its KVC and r_j has reached the point b tokens before r_i's KVC space. b functions as a buffer to handle the inaccurate estimation and it is set to a small number (e.g., 8 tokens). Therefore, if $a_j - (u_j + s_i^o) - (s_i^p + s_i^o) \geq b$, it means that we can allocate r_i in the allocated KVC of r_j starting from the red line $s_0 = a_j - (u_j + s_i^o) - b$. To choose a request to embed request r_i, we choose the running request r_j that has the minimum remaining allocated KVC in order to reduce reserved waste. Due to inaccurate estimation of s_i^o, if r_i does not complete by the time when r_j needs the KVC allocated to r_i, then r_i must be preempted. The value of b is empirically determined; a larger b leads to higher reserved waste while a smaller b may cause the preemption of r_i.

Request Selection and KVC Allocation

The requests in a returned batch after an iteration are called *returned requests*. In forming a new batch and allocating KVC, we make sure that (1) the waiting requests' SLO_{ttft} and the returned requests' SLO_{tbt} must be satisfied, and (2) we allocate the KVC to a request as close to its demand as possible to avoid preemptions to reduce TBTs while reducing reserved waste to reduce both TTFTs.

The waiting requests are ordered by the ascending order of their Remaining Time (RT) to their SLO_{ttft} and the SLO_{tbt} for preempted requests, denoted by RT_{ttft} and RT_{tbt}. The requests are sequentially selected to add to the batch. We use T_I^{max} to denote the maximum iteration latency historically. Then, the waiting requests that have $(\text{RT}_{ttft} - \text{T}_\text{I}^{max}) < \epsilon$, where ϵ is a small number, must be batched in order to satisfy their SLO_{ttft}. We use $\mathbf{N}_w$ to denote the group of such waiting requests, and $\mathbf{N}_r$ the group of returned requests that have used up their allocated KVC and must execute in the next iteration to satisfy their SLO_{tbt}, that is, $\text{RT}_{tbt} - \text{T}_\text{I}^{max} < \epsilon$. The requests in $\mathbf{N}_w$ and $\mathbf{N}_r$ are called *critical requests*, which need to allocate KVC first. Below, we present how to allocte KVC to critical requests and then to other returned requests and requests selected to exhaust the token budget (named as non-critical requests).

(1) Allocate KVC to critical requests. We define a small block (e.g., 8 blocks), denoted by B. The basic KVC requirement of a critical waiting request equals to the sum of its prompt length and a block size: $(s_i^p + B)$, and that of a critical returned request equals to B. The total basic KVC requirement of critical requests equals to $D_{kvc} = \sum_i^{|\mathbf{N}_w|}(s_i^p + B) + \sum_i |\mathbf{N}_r|B$. We sort the critical requests in descending order of their basic requirements and allocate each request using the embedding method first. Then, we allocate the remaining critical requests to unallocated KVC. If the current unallocated KVC $A'_{kvc} < D_{kvc}$,

preemptions are executed to make $A'_{kvc} = D_{kvc}$. Which requests to choose for preemptions is introduced in 5.3.4. When $A'_{kvc} \geq D_{kvc}$, we allocate the basic required KVC to each remaining critical request. Then, if there remains unallocated KVC, we allocate it to all critical requests and other non-critical requests The details are presented below.

(2) Allocating remaining KVC for the next iteration. We use $\mathbf{N}'_r$ to denote the group of returned requests that are not critical. In addition to critical requests, $\mathbf{N}'_r$, we sequentially select requests from the waiting requests to use up the token budget (denoted by $\mathbf{N}'_w$) to minimize waiting time and maximize throughput. All non-critical requests ($\mathbf{N}'_w$ and $\mathbf{N}'_r$) and critical requests ($\mathbf{N}_w$ and $\mathbf{N}_r$) will join the KVC allocation. We use n to denote the total number of these requests. The next question is how to distribute A'_{kvc} among these requests. In this step, we allocate KVC to each request to match its total demand ($M_i = S_i^p + S_i^o$) as closely as possible while accommodating the n requests to the batch.

For non-critical requests, we use the embedding method first. If it successfully allocates a request, the request's M_i becomes 0 . Then, we allocate A'_{kvc} to the remaining requests for the following three cases:

- When $\sum_{i=1}^{n} M_i = A'_{kvc}$, the requests are added to the batch and are guaranteed to receive the KVC they demand during execution.
- When $\sum_{i=1}^{n} M_i < A'_{kvc}$, adding more requests can more fully allocate the KVC, increase dequeuing speed and increase throughput slightly, but may also increase iteration time. The iteration time, estimated based on profiled data, increases almost linearly with the number of tokens in a batch [1]. Therefore, more requests are sequentially added from the queue until $\sum_{i=1}^{n} M_i = A'_{kvc}$ or the SLO of any request in the batch is violated.
- When $\sum_{i=1}^{n} M_i > A'_{kvc}$, we amortize the excess demands among the requests, and allow a request to use the released KVC from another request once the latter exhausts its allocated KVC. A request with a looser SLO and with a longer predicted output length should have a higher KVC demand cut since it is less delay-sensitive and also have more opportunities to receive KVC. Therefore, we incorporate these two factors in the amortization process. The weight for each request is computed as: $w_i = \frac{\text{RT}_i}{\sum_i \text{RT}_i} \times \frac{s_i^p}{\sum_i s_i^p}$. Subsequently, the KVC allocated to each request is determined by: $A_i = A'_{kvc} \times \frac{w_i}{\sum_{i=1}^{n} w_i}$.
  After the above amortization, a request r_i may not receive its demanded KVC fully. To address this, we find a running request r_j that will release its KVC shortely before r_i exhausts its allocated KVC, and the released KVC is no less than r_i's additional demand. Then, r_i will use r_j's released KVC.

Proactive KVC Allocation and Global Reservation

A request may not receive its KVC demand M_i (an unfulfilled request) and then if it exhausts its allocated KVC without any request completing, a preemption occurs. The proposed proactive KVC allocation and global reservation aim to avoid the preemptions. In the previous systems, completed requests' released KVC is used to accommodate requests from the waiting queue. To avoid preemptions, after allocating the KVC to the critical requests, we include unfulfilled running requests, predicted to complete within m iterations (where m is a small number), into the non-critical request group for the remaining KVC allocation.

Due to the autoregressive nature of LLMs and the imprecision in output length estimation, a request may not receive the necessary KVC when needed. Assigning additional padding to each request can lead to wasted reserved memory that may go unused. To address this, we reserve a shared KVC space for all requests, allowing them to use the space in case of KVC allocation failure.

5.3.4 Preemption Policy

The preemption policy determines the order for requests to be preempted to reduce both the number and duration of preemptions. Based on O5.3, CacheOPT considers three factors: 1) TBT SLO, 2) SRTF, and 3) LKVO. The TBT SLO of a request needs to be satisfied so we choose a loose-SLO request to be preempted. The reasons for considering SRTF and LKVO are explained in 5.2.4. The remaining processing time is measured by the remaining predicted output tokens. For each factor, we set up certain magnitude ranges and order requests accordingly. For example, 0.05-0.2s, 0.2-0.5s, and 0.5-2s for the TBT SLO, and 0-128, 128-256, 256-384, 384-512, ... in tokens for the remaining processing time and occupied KV-cache. The requests first are ordered based on the descending order of TBT SLOs. Then, for the requests with similar TBT SLOs, they are ordered in the descending order of remaining processing time. Next, for the requests with similar remaining processing time, they are ordered in the ascending order of the KVC occupancy. Finally, CacheOPT preempts the first request.

5.3.5 Preemption Strategy Selection

As per O5.4, if the sequence length is greater than a sweet spot, swapping is more time-efficient than recomputation. The key is to find this sweet spot. For this purpose, we can profiling and regression models for estimation.

Profiling is conducted using the LLM system with the specific settings including the type of GPUs, the number of GPUs, the tensor parallelism (TP) and model parallelism (MP) degrees and the model. For a given sequence length S, total memory bandwidth M_b and total GPU capacity G, of the target hardware

(e.g., A100), we measure the recomputation time for a range of sequence lengths, and use the data to train a polynomial regressor:

$$L_r(S) = \alpha_r \cdot S^{\beta_r} + \kappa_r \cdot S + \epsilon_r,$$

where α_r, β_r, κ_r and ϵ_r are parameters determined during training. We choose the polynomial regressor because reecomputation latency typically involves complex operations like matrix multiplications and other non-linear computations that grow in polynomial complexity with the size of the input [78].

Similarly, we profile swapping times across various sequence lengths and use the results to train a linear regressor.:

$$L_s(S, M_b, G) = \gamma_s \cdot S + \delta_s,$$

where γ_s and δ_s are determined during training. We choose the linear regressor because swapping latency linearly increases based on the data size [83]. We calculate the sweet spot S_s, defined as the sequence length satisfying $L_r(S) = L_s(S)$.

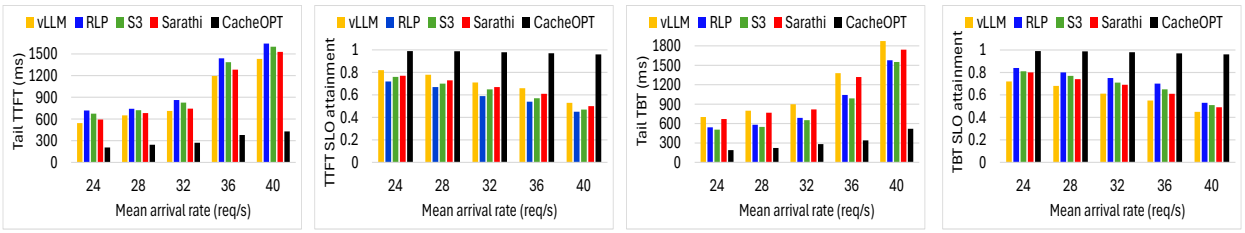
5.4 Implementation

We implemented CacheOPT based on the vLLM source code [60], comprising about 6K lines of Python code. To integrate confidence-based padding, we modified the *scheduler.py* file. Specifically, we added a new function, *compute_confidence_padding* and extended the *allocate_kvc* function in vLLM to incorporate this padding method.

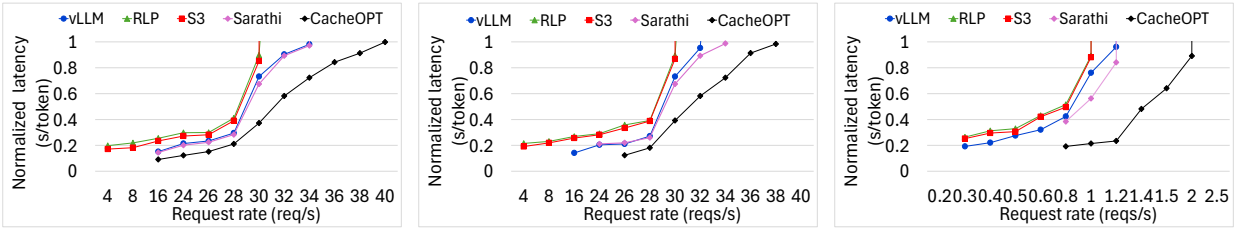
We modified *kvcache.py* to include our *embedding method*. We added a function named *form_batch* in the *scheduler.py*. We further added functionalities in the *allocate_kvc* function to allocate KVC. New function *reuse_unused_kvc* was integrated into the *TokenBlockManager* class in vLLM to reallocate unused KVC. We also added function *proactive_allocation* in *scheduler.py* to proactively allocate KVC.

We added another function *global_reservation* that sets up a reserved portion of the KVC. To track reserved KVC, in the *TokenBlockManager* class in *kvcache.py*, we added a function named *track_reserved_blocks* to manage the tracking, allocation, and release of reserved blocks. This function also ensures that the reserved blocks are separate from the general allocation pool, preventing conflicts during allocation. Function *allocate_reserved_kvc* is added in *scheduler.py* to assign reserved KV-cache blocks to requests.

We replaced the default FCFS preemption policy in the *evict_request* function in vLLM with our preemption policy. Besides, we added a function, *preempt_request_strategy*, which decides the preemption strategy.



(a) Tail TTFT (b) TTFT SLO attainment (c) Tail TBT (d) TBT SLO attainment



(e) Alpaca. (f) ShareGPT (g) Bookcorpus

Figure 5.12: End-to-end latency performance for OPT-13B.

5.5 Performance Evaluation

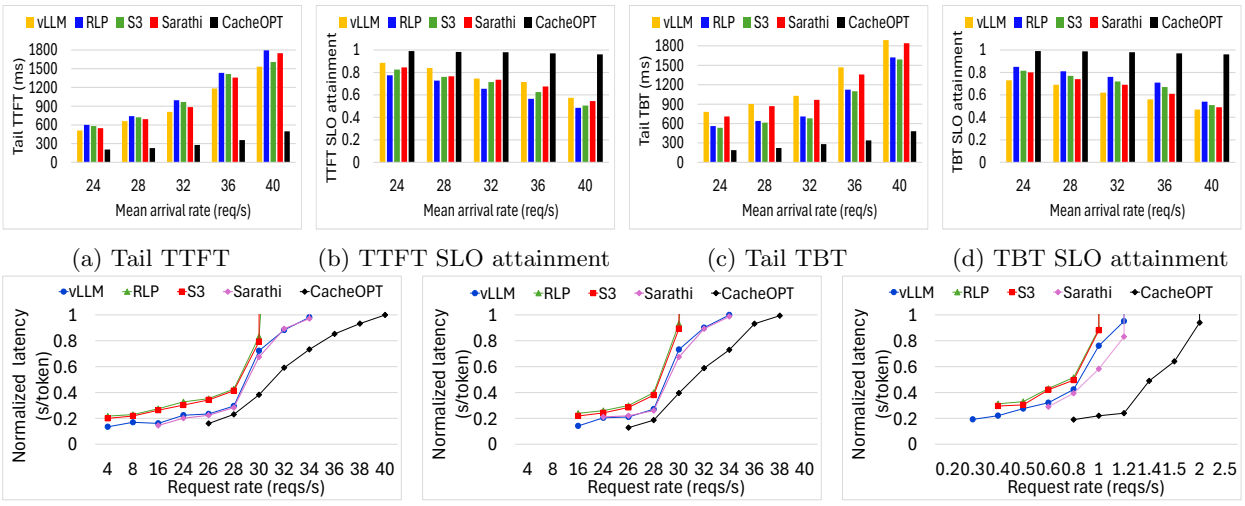
Experiment Settings. Unless otherwise specified, the experiment settings are the same as in 5.2. For setting the TTFT and TBT SLO, we measured the average TTFT and TBT across all compared methods and multiply them by a SLO scale. The SLO scale was randomly selected from [0.5, 2.5]. Furthermore, to make the TTFT SLO reasonable for a long prompt that needs chunking [76], we increased its TTFT SLO by multiplying it with a scaling factor equal to the number of chunks. We used 3-hour trace for OPT-13B and used 1-hour trace for OPT-175B. By default, we set the block size (B) to 8 tokens, the number of preallocation iterations m to 2, the number of reserved blocks to 8, the confidence to 90%, $\alpha = 8$, and $\beta = 100$, respectively.

Output length predictor. We used 9K requests and 6k for inference for testing the performance. The fine-tuning process took around 9 hours.

Compared Methods. We compared CacheOPT with vLLM, RLP, S^3, Sarathi-Serve (Sarathi for simplicity). Due to space limit, unless otherwise specified, we plot the average of the results across three datasets for each model.

5.5.1 Overall Performance Comparison

TTFT and TBT. Figures 5.12a, and 5.13aillustrate the *tail TTFT* during the entire running time versus the mean request arrival rate for different models. Figures 5.12b, and 5.13b,show the *TTFT SLO attainment*, which indicates the fraction of requests that meet their TTFT SLOs. We observe that the tail TTFT keeps



(a) Tail TTFT (b) TTFT SLO attainment (c) Tail TBT (d) TBT SLO attainment

(e) Alpaca (f) ShareGPT (g) Bookcorpus

Figure 5.13: End-to-end latency performance for OPT-175B.

increasing and the TTFT SLO attainment keep decreasing as the arrival rate increases. While the compared methods exhibit abrupt changes, CacheOPT demonstrates a steady progression. CacheOPT reduces the tail TTFT of vLLM, RLP, S^3, and Sarathi by 64%-2.34×, 1.21×-2.83×, 1.11×-2.73×, and 97%-2.56× across all arrival rates and models, respectively. Following a similar TTFT trend, CacheOPT achieves 0.94-0.97 TTFT SLO attainment, with the highest improvement of 47% over Sarathi, the best compared system, at 40 req/s. This improvement stems from advanced methods that mitigate KVC bottlenecks to enable more requests per batch while reducing both the number and duration of preemptions, ultimately decreasing waiting time. The *confidence-based padding* method reduces preemptions, while the *embedding method* mitigate the KVC bottleneck. *Request selection and KVC allocation* tries to limit the TTFT. *Proactive KVC allocation* and *global KVC reservation* further reduce the preemptions. Additionally, CacheOPT's advanced preemption policy reduces the number and duration of preemptions. Further, *preemption strategy selection* minimizes swapping or recomputation time, allowing waiting requests to be scheduled earlier and reducing TTFT. Overall, the improvement for large models is smaller compared to small models due to reduced KVC competition when using more GPUs.

Figures 5.12c,and 5.13c illustrate the *tail TBT* during the entire experiment. Figures 5.12d, and 5.13d show the *TBT SLO attainment*, which indicates the fraction of requests whose all iterations meet their TBT SLOs. Similarly, as the arrival rate increases, while the compared methods exhibit abrupt changes, CacheOPT demonstrates a steady progression. CacheOPT achieves the lowest tail TBT among all systems. Specifically, CacheOPT reduces the tail TBT of vLLM, RLP, S^3, and Sarathi by 1.82-3.29×, 1.42-2.37×, 1.31-2.30×, and 1.47-2.71× across all arrival rates and models. CacheOPT achieves 0.94-0.98 TBT SLO attainment,

outperforming Sarathi by 53% at 40 req/s. The superior performance of CacheOPT can be attributed to its advanced strategies explained above. Mitigating KVC competition also reduces preemptions. The reduction of the number and duration of preemptions directly reduces TBT and avoid TBT SLO violations. Considering TBT SLO in selecting requests in batching and for preemptions further reduce TBT and avoid TBT SLO violations.

Normalized latency. *Normalized latency* is a request's end-to-end latency divided by its output length [135, 60]. Figures 5.12e-5.12g, and 5.13e-5.13g show the average normalized latency of the systems versus the mean arrival rate. A high-throughput serving system should retain low normalized latency against high request rates. We compare the request rates that the systems can sustain while maintaining similar latencies. CacheOPT can sustain 1.29-1.89× higher arrival rates than vLLM, 1.44-1.94× than RLP and S^3, and 1.24-1.58× than Sarathi on average for the three datasets. CacheOPT's advantages on BookCorpus is more pronounced because it contains longer sequences, allowing fewer requests to be batched and generating more KVC competition.

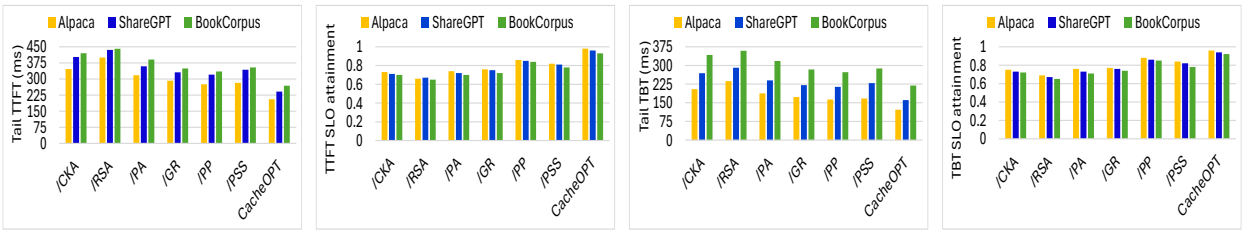
Similar to the tail TTFT, for other metrics, CacheOPT shows lower improvement for larger models compared to smaller models and as arrival rates increase, CacheOPT demonstrates greater improvements over other systems due to the same reasons.

5.5.2 Ablation Study

We tested the following variants of CacheOPT to evaluate the effectiveness of its components.

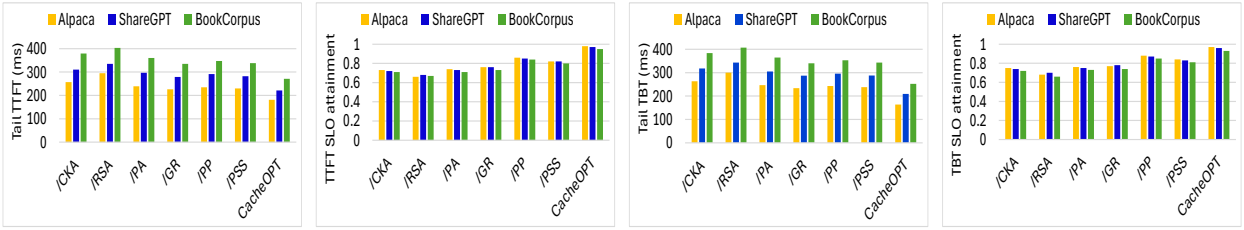
- */CKA*: CacheOPT without Confident-based KVC Allocation and it adds a fixed padding of 10% of the predicted output length.
- */RSA:* CacheOPT without Request Selection and KVC Allocation and it uses FCFS for request selction and allocates KVC using RLP.
- */PA:* CacheOPT without Proactive KVC allocation.
- */GR:* CacheOPT without Global Reservation
- */PP:* CacheOPT without Preemption Policy and it follows FCFS.
- */PSS:* CacheOPT without Preemption Strategy Selection and its uses the default recomputation in the vLLM code.

Figures 5.14, and 5.15 show the different metrics of the variants in the four models. We now discuss the average results across the four models. CacheOPT reduces the tail TTFT by 35%, 44%, 29%, 24%, 27%, 28% compared to /CKA, /RSA, /PA, /GR, /PP, and /PSS, respectively. For TTFT SLO attainment, these



(a) Tail TTFT (b) TTFT SLO attainment (c) Tail TBT (d) TBT SLO attainment

Figure 5.14: Ablation study for OPT-13B.



(a) Tail TTFT (b) TTFT SLO attainment (c) Tail TBT (d) TBT SLO attainment

Figure 5.15: Ablation study for OPT-175B.

variants exhibit degradation of 27%, 34%, 26%, 23%, 14%, and 17%, respectively. For TBT, CacheOPT demonstrates improvements of 40%, 48%, 35%, 29%, 25% and 27% compared to these variants, respectively. TBT SLO attainment for these variants shows degradations of 22%, 28%, 21%, 20%, 8%, and 13%, respectively. This demonstrates that removing or replacing critical design elements leads to performance degradation, and the critical role of all design components in reducing tail TTFT and TBT, and SLO violations.

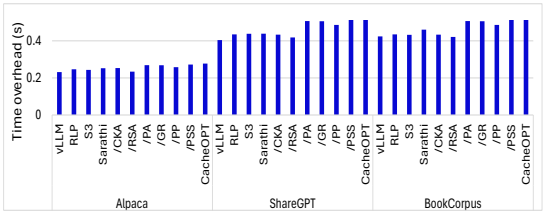


Figure 5.16: Scheduling time overhead.

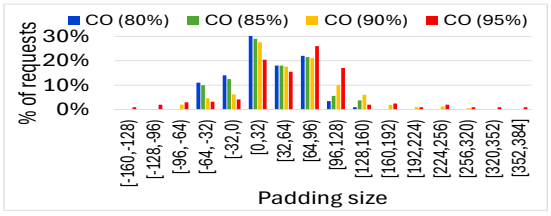
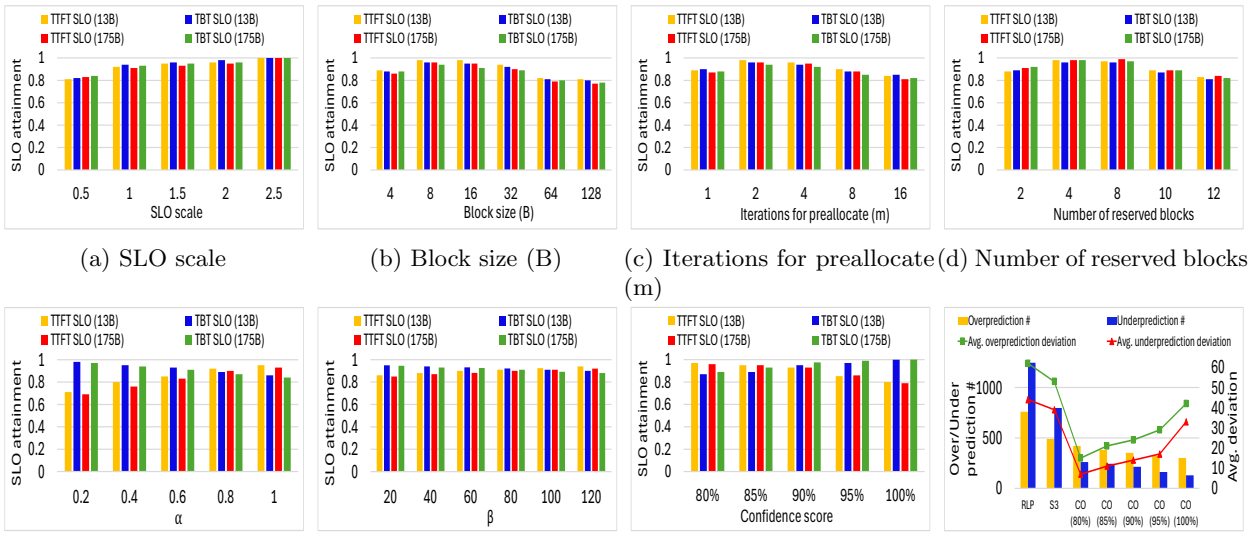


Figure 5.17: Padding size.

5.5.3 Time Overhead

Figure 5.16 shows the time overhead for different systems and for different variants of CacheOPT in each iteration. CacheOPT have 80%, 77%, 18%, and 23% higher time overhead than vLLM, Sarathi, RLP and S^3 respectively. These higher time overheads only constitute 0.003% over the iteration. The overhead of the components of the CacheOPT contributes to its time overhead. Specifically, compared to the CacheOPT, /CKA, /RSA, /PA, /GR, /PP, /PPS show 17%, 41%, 3% , 3.14%, 7%, 1.8% less time overhead, indicating the time overhead of each component.



(a) SLO scale (b) Block size (B) (c) Iterations for preallocate (m) (d) Number of reserved blocks

(e) α (f) β (g) Confidence score. (h) Response length predictor

Figure 5.18: Sensitivity testing.

5.5.4 Sensitivity Testing

Figure 5.18a shows the SLO attainment for the two OPT models, versus the SLO scale. The figure show a steady increase in SLO attainment as the scale increases. At the smallest scale of 0.5, CacheOPT still can achieve around 0.85 TTFT and TBT SLO attainments. As the scale increases to 1.5, the SLO attainments surpass 0.95, and they reach nearly 1.0 at the scale of 2.5. The results verify the capability of CacheOPT in providing high TTFT and TBT SLO attainments, and good user experience.

Figures 5.18b shows the SLO attainments versus the block size (B). As the block size increases from 4 to 8, the SLO attainment increases, then as the block size increases, the SLO attainments gradually decrease. A smaller block size (e.g., 4) results in underprovisioning, which also leads to high TBT and hence high TTFT, though they do not increase significantly. This is because requests with insufficient KVC can still obtain additional KVC proactively or on demand. Conversely, a larger block size causes overprovisioning, reducing the number of requests per batch and increasing TTFT and KVC competition and hence TBT. Experimental results indicate that a block size of 8 performs best in our setup.

Figure 5.18c shows the SLO attainments versus the number of iterations for preallocation (m). At $m = 1$, SLO attainments are around 0.85, peaking at 1.0 when $m = 2$, before gradually declining beyond $m = 2$ due to reserved waste. Hence, $m = 2$ is the optimal setting in our experimental setup.

Figure 5.18d shows the SLO attainments versus the number of reserved blocks. The SLO attainments increase as the number of reserved blocks increases from 2 to 4 and then to 8, but then decrease when it keeps decreasing. Insufficient reserved KVC cannot satisfy KVC demands from some requests but high reservation increases reserved waste. Therefore, 4 and 8 are suitable number of reserved blocks in our experiment settings.

Figure 5.18e shows the impact of the variable α in Equation (5.4), which controls the Hoeffding's inequality for CacheOPT with β. For $\beta = 100$, with the increase of α, we observe that the TTFT SLO keeps increasing, but the TBT SLO keeps decreasing. This happens because increasing α increases c_i, which increases padding. Figure 5.18f shows the same plot for β, for $\alpha = 0.9$. For increasing β, we observe that TTFT SLO keeps decreasing while the TBT SLO keeps increasing, because increasing β decreases c_i, which decreases padding.

Figure 5.18g shows the impact of confidence score. Lower confidence scores (e.g., 80%) lead to smaller positive and negative padding sizes. Smaller positive padding sizes enable more requests accommodated in a batch, which reduces waiting time hence TTFT but increases preemptions hence TBT. On the other hand, smaller negative padding sizes reduce the number of requests accommodated in a batch, which increases waiting time but decreases preemptions. Our output length predictor results in more underpredictions than overpredictions, as shown in Figure 5.17, which illustrates the distribution of padding sizes across the requests. Therefore, the influence from positive paddings dominates that from negative paddings. As a result, there is a trend of increasing TBT SLO attainments and decreasing TTFT SLO attainments as the confidence score increases. At the confidence score of 0.9, both TTFT and TBT SLO attainments exceed 0.9, striking a balance between TTFT and TBT.

Figures 5.18h shows the impact of the confidence score on the number of overpredictions and underpredictions, along with their average deviations. We also include those of RLP and S^3 as reference. Compared to RLP and S^3, CacheOPT exhibits significantly fewer overprovisions, underprovisions, and deviations, demonstrating its high accuracy in output length estimation. As the confidence score increases, the number of overprovisions and underprovisions decreases, but their average deviations increase. This occurs because larger padding sizes reduce misestimations but increase deviation magnitude. The results emphasize the need for an optimal confidence score.

Overall, these results demonstrate the adaptability of CacheOPT across a wide range of configurations. By effectively adapting KV cache allocation with workload demands, CacheOPT maintains high SLO attainments, ensuring reliable performance under diverse conditions.

5.6 Related Work

Orca [135] uses an iteration-level scheduling strategy combined with maximum resource allocation, which results in under-utilization of GPU resources. To address this inefficiency, vLLM [60] introduces a block-based allocation strategy, and prediction-based KVC allocation methods [49, 146] predict output lengths and allocate the KVC equal to the predicted size. Sarathi-Serve [2] incorporates chunked-prefills and stall-free scheduling to address long sequences. FastServe [123] utilizes preemptive scheduling to minimize JCT. Llumnix [106] reschedules requests across multiple model instances, effectively reducing tail latencies. vAttention [89] is a memory management system that uses virtual memory to store and manage the KV cache. Cheng *et al.* [15] developed a scheduling method that splits long-generation tasks into smaller parts, to make it easier to manage resources and predict serving times. ALISA [143] combines a Sparse Window Attention algorithm to reduce the memory footprint of KVC with three-phase token-level dynamic scheduling to optimize the balance between caching and recomputation. Several approaches focus on resource optimization to improve LLM performance. ExeGPT [85] optimizes resource usage and adjusts execution settings like batch size and parallelism. INFERMAX [54] analyzes different scheduling strategies, focusing on balancing costs and resource usage. Sheng *et al.* [103] focused on achieving fairness in scheduling. Lee *et al.* [63] proposed prefetching only the essential KV cache entries for computing the subsequent attention layer. DistServe [147] decouples the prefill and decode phases, running them on separate machines or GPUs. Some other methods [122, 69] aim to enhance the efficiency and performance of LLM applications either by optimizing application-level operations or by dynamic adapter management targeting LoRA. Unlike previous work, we study the impact of the allocated KVC amount on the tradeoff between satisfying the TTFT and TBT SLOs, and propose novel methods to maximize the attainment of both TTFT and TBT SLOs.

5.7 Conclusion

Our proposed CacheOPT addresses the challenge to satisfy both TTFT and TBT SLOs. It incorporates four components: 1) confidence-based padding, 2) SLO-aware batching and KVC allocation, 3) preemption policy, and 4) preemption strategy selection. Experimental results show that CacheOPT achieves up to a 3.29× and 2.83× lower tail TBT and TTFT, and 47% and 53% higher TTFT and TBT SLO attainment than the state-of-the-art methods. In the future, we will design methods to automatically determine the optimal parameters in CacheOPT.

Chapter 6

Conclusion

The rapid advancement of deep learning and edge device technology significantly enhances our everyday lives. However, the association between both brings forth challenges that hinder the process of efficient training and inference on these devices. This dissertation outlines and tackles three key challenges associated with efficient training and inference on edge devices. They are– (1) resource constraints of the edge devices, (2) excessive load on one device, (3) variation of performance among the edge accelerators, and (4) LLM KVC management for edge devices. These challenges are examined and tackled within the realms of emerging technologies in this dissertation and lead to significant contributions.

The first contribution is the introduction of a distributed training system called DMP. This system leverages Data and Model Parallelism to optimize the training structure by clustering edge devices. By exploiting geographically close nodes for data sensing and partitioning model tasks, DMP reduces overall training time while maintaining accuracy.

Another significant contribution is the development of SROLE, a decentralized scheduling system utilizing Shielded RL to mitigate resource overloading and action collisions in data and model parallel training scenarios. SROLE empowers each edge node to autonomously schedule jobs, thereby enhancing efficiency and reducing training time.

Furthermore, the dissertation proposes a system named FLEX for Fast, Accurate DNN Inference on Low-Cost Edges. This system dynamically determines layer assignments across CPU and accelerator using heuristic and RL-based mechanisms. This system aims to optimize inference performance while considering the resource constraints and energy efficiency of low-cost edge devices.

Finally, the dissertation proposes a system named CacheOPT, that enhances KV-cache (KVC) management for LLM inference by using demand-based KVC allocation. It incorporates confidence-aware padding to fine-tune memory allocation dynamically, minimizing under- and over-provisioning, and uses advanced preemption strategies, including a cost-based heuristic to choose between recomputation and swapping, to optimize response latency. Together, these techniques improve memory efficiency and reduce latency, making LLM deployment more feasible in constrained environments.

Overall, the dissertation's contributions underscore the importance of addressing the unique challenges posed by edge computing in DL training and inference. By leveraging systematic heuristic and RL-based approaches, the proposed systems demonstrate significant improvements in performance metrics such as time, accuracy, and energy efficiency. These advancements hold promise for revolutionizing edge-based machine learning applications, making them more accessible, efficient, and scalable for diverse use cases in real-world scenarios.

Two immediate open questions require further investigation and exploration in the future. One concern is the scalability and robustness of the DL training and inference on edge devices. We will explore techniques for dynamic adaptation to changing network conditions and mobility, fault tolerance mechanisms, and optimized resource allocation strategies.

Finally, to fully evaluate the effectiveness of our system, CacheOPT on real-world edge devices, it is essential to conduct performance measurements of the system on actual hardware setups. Testing on real edge devices will provide insights into practical challenges like memory limitations, processing delays, and the variability in device capabilities, ensuring our system is robust and optimized for these environments. Additionally, we aim to explore input-oriented KVC sharing, which would allow multiple requests with similar inputs to share key-value cache entries efficiently. This approach could further reduce memory usage and response latency by avoiding redundant storage of similar data across requests, ultimately enhancing the scalability and efficiency of large language model inference on edge devices.

Bibliography

[1] Amey Agrawal et al. "StableGen: Efficient LLM Inference with Low Tail Latency". In: *Proc. of OSDI* (2024).

[2] Amey Agrawal et al. "Taming {Throughput-Latency} Tradeoff in {LLM} Inference with {Sarathi-Serve}". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 117–134.

[3] *Air Quality Data in India*. https://www.kaggle.com/rohanrao/air-quality-data-in-india.

[4] Android. *Native Development Kit*. https://developer.android.com/ndk. 2020.

[5] D. Anguita et al. "A Public Domain Dataset for Human Activity Recognition using Smartphones". In: *The European Symposium on Artificial Neural Networks*. 2013.

[6] Anubhav Ashok et al. "N2N Learning: Network to Network Compression via Policy Gradient Reinforcement Learning". In: *Proc. of ICLR*. 2018.

[7] Yoshua Bengio et al. "Curriculum learning". In: *International Conference on Machine Learning*. 2009.

[8] Vidmantas Bentkus. "On Hoeffding's inequalities". In: *Annals of probability* (2004), pp. 1650–1673.

[9] George Boateng et al. "Experience: Design, Development and Evaluation of a Wearable Device for mHealth Applications". In: *Proc. of MobiCom*. 2019.

[10] Keith Bonawitz et al. "Towards Federated Learning at Scale: System Design". In: *Proc. of SysML*. 2019.

[11] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[12] Jason Brownlee. *Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras*. https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-\python-keras/. 2019.

[13] Jacky Cao et al. "Mobile augmented reality: User interfaces, frameworks, and intelligence". In: *ACM Computing Surveys (CSUR)* (2021).

[14] Yitao Chen et al. "Exploring the Use of Synthetic Gradients for Distributed Deep Learning across Cloud and Edge Resources". In: *Proc. of HotEdge*. 2019.

[15] Ke Cheng et al. "Slice-Level Scheduling for High Throughput and Load Balanced LLM Serving". In: *ArXiv* abs/2406.13511 (2024). URL: https://api.semanticscholar.org/CorpusID:270619678.

[16] Yu Cheng et al. "Model compression and acceleration for deep neural networks: The principles, progress, and challenges". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136.

[17] Tomas Chladek. *Profiler*. https://play.google.com/store/apps/details?id=cz.chladek.profiler&hl=en_US&gl=US.

[18] François Chollet et al. *Keras*. https://keras.io. 2015.

[19] Sai-Ho Chung. "Applications of smart technologies in logistics and transport: A review". In: *Transportation Research Part E: Logistics and Transportation Review* 153 (2021), p. 102455.

[20] ARM Community. *Benchmarking floating-point precision in mobile GPUs*. https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/benchmarking-floating-point-precision-in-mobile-gpus. Accessed: 09-12-2022.

[21] Elias Coninck et al. "Distributed Neural Networks for Internet of Things: The Big-Little Approach". In: vol. 170. Nov. 2016.

[22] *cpulimit*. https://github.com/opsengine/cpulimit.

[23] Aidan Curtis et al. "HealthSense: Software-defined Mobile-based Clinical Trials". In: *Proc. of MobiCom*. 2019.

[24] *Damaged cell towers create communication problems in Northern California fire zone*. http://www.latimes.com/local/cali northern-california-fires-live-damaged-cell-towers-create-1507667633-htmlstory.html#. Accessed in: March, 2021.

[25] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[26] Digibites. *AccuBattery*. https://play.google.com/store/apps/details?id=com.digibites.accubattery&hl=en&gl=US&pli=1. Online; accessed 11 January 2023.

[27] Ingy ElSayed-Aly et al. "Safe Multi-Agent Reinforcement Learning via Shielding". In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. 2021.

[28] *EMNIST*. http://cis.jhu.edu/~sachin/digit/digit.html.

[29] A. Fan et al. "Training with Quantization Noise for Extreme Model Compression". In: *Proc. of ICLR*. 2021.

[30] J. Fan et al. "Deadline-Aware Task Scheduling in a Tiered IoT Infrastructure". In: *Proc. of GLOBECOM*. 2017.

[31] Biyi Fang, Xiao Zheng, and Mi Zhang. "NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision". In: *Proc. of MobiCom* (2016).

[32] Oscar Ferraz et al. "Benchmarking Vulkan vs OpenGL Rendering on Low-Power Edge GPUs". In: *2021 International Conference on Graphics and Interaction (ICGI)*. IEEE. 2021, pp. 1–8.

[33] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560 (2018).

[34] Robert Geirhos et al. "Partial success in closing the gap between human and machine vision". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 23885–23899.

[35] GSMA. *The Mobile Economy 2023*. https://www.gsma.com/mobileeconomy/wp-content/uploads/2023/03/270223-The-Mobile-Economy-2023.pdf. 2023.

[36] Jashwant Raj Gunasekaran et al. "Cocktail: A Multidimensional Optimization for Model Serving in Cloud". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022.

[37] Peizhen Guo, Bo Hu, and Wenjun Hu. "Mistify: Automating DNN Model Porting for On-Device Inference at the Edge". In: *NSDI*. 2021.

[38] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *ICML*. 2018.

[39] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *Proc. of ICLR*. 2015.

[40] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. "Distributed deep learning on edge-devices: feasibility via adaptive compression". In: *Proc. of NCA*. 2017.

[41] S. Hardy et al. "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption". In: *ArXiv* abs/1711.10677 (2017).

[42] Yann-Michaël De Hauwere, Peter Vrancx, and Ann Nowé. "Learning multi-agent state space representations". In: *Adaptive Agents and Multi-Agent Systems*. 2010. URL: https://api.semanticscholar.org/CorpusID:11652380.

[43] Wassily Hoeffding. "Probability Inequalities for Sums of Bounded Random Variables". In: *Journal of the American Statistical Association* 58.301 (1963), pp. 13–30.

[44] S. Huang et al. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis". In: *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 2010.

[45] Yutao Huang et al. "When deep learning meets edge computing". In: *Proc. of ICNP*. 2017.

[46] *Is Your Data Center Prepared for a Natural Disaster?* https://www.vxchnge.com/blog/data-center-prepared-for-natural-disaster.

[47] Hyuk-Jin Jeong et al. "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers". In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2018, pp. 401–411.

[48] Yimin Jiang et al. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *Proc. of OSDI*. 2020.

[49] Yunho Jin et al. "S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput". In: *arXiv preprint arXiv:2306.06000* (2023).

[50] Yunho Jin et al. "S3: increasing GPU utilization during generative inference for higher throughput". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. 2024.

[51] Manju Lata Joshi and Nehal Kanoongo. "Depression detection using emotional artificial intelligence and machine learning: a closer review". In: *Materials Today: Proceedings* 58 (2022), pp. 217–226.

[52] Woosung Kang et al. "LaLaRAND: Flexible Layer-by-Layer CPU/GPU Scheduling for Real-Time DNN Tasks". In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021.

[53] *Keras CNN*. https://keras.io/examples/mnist_cnn/.

[54] Kyoungmin Kim et al. "The Effect of Scheduling and Preemption on the Efficiency of LLM Inference Serving". In: 2024. URL: https://api.semanticscholar.org/CorpusID:273969695.

[55] Youngsok Kim et al. "Layer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019.

[56] Sosuke Kobayashi. *Homemade BookCorpus*. https://github.com/soskek/bookcorpus. 2018.

[57] Vijay R Konda and John N Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[58] H. W. Kuhn and Bryn Yaw. "The Hungarian method for the assignment problem". In: *Naval Res. Logist. Quart* (1955), pp. 83–97.

[59] Aditya Kusupati et al. "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network". In: *Proc. of NIPS*. 2018, pp. 9017–9028.

[60] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles* (2023). URL: https://api.semanticscholar.org/CorpusID:261697361.

[61] Liangzhen Lai and Naveen Suda. "Enabling deep learning at the IoT edge". In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 135.

[62] Averill M Law, W David Kelton, and W David Kelton. *Simulation modeling and analysis*. Vol. 3. McGraw-Hill New York, 2000.

[63] Wonbeom Lee et al. "InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 155–172. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/lee.

[64] Changlin Li et al. "Block-wisely Supervised Neural Architecture Search with Knowledge Distillation". In: *Proc. of CVPR*. 2020, pp. 1989–1998.

[65] Guangli Li et al. "Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge". In: *International Conference on Artificial Neural Networks*. Springer. 2018, pp. 402–411.

[66] He Li, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: Deep learning for the Internet of Things with edge computing". In: *IEEE Network* 32.1 (2018), pp. 96–101.

[67] Qinbin Li, Zeyi Wen, and Bingsheng He. "Practical Federated Gradient Boosting Decision Trees". In: *Proc. of AAAI*. 2019.

[68] Petro Liashchynskyi and Pavlo Liashchynskyi. "Grid search, random search, genetic algorithm: a big comparison for NAS". In: *arXiv preprint arXiv:1912.06059* (2019).

[69] Chaofan Lin et al. "Parrot: Efficient Serving of LLM-based Applications with Semantic Variable". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 929–945. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

[70] Sicong Liu et al. "On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework". In: *Proc. of MobiSys*. 2018.

[71] Y. Long et al. "Complexity-aware Adaptive Training and Inference for Edge-Cloud Distributed AI Systems". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021.

[72] Andrey Malinin, Bruno Mlodozeniec, and Mark Gales. "Ensemble distribution distillation". In: *Proc. of ICLR*. 2020.

[73] J. Masters. "A look back at the horrific 2020 atlantic hurricane season. yale climate connections". In: (Accessed in: March, 2021). https://yaleclimateconnections.org/2020/12/a-look-back-at-the-horrific-2020-atlantic-hurricanecenter/, Accessed in: March, 2021.

[74] H. B. Mcmahan et al. "Communication-efficient learning of deep networks from decentralized data". In: *Proc. of ICAIS*. 2017.

[75] Paulius Micikevicius et al. "Mixed Precision Training". In: *Proc. of ICLR* (2018).

[76] Microsoft. *DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference*. https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen. Online; accessed in April 2024.

[77] Jayashree Mohan et al. "Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022.

[78] Deepak Narayanan et al. "Efficient large-scale language model training on gpu clusters using megatron-lm". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[79] Deepak Narayanan et al. "PipeDream: Generalized Pipeline Parallelism for DNN Training". In: *Proc. of SOSP*. 2019.

[80] Vladimir Nekrasov et al. "Fast neural architecture search of compact semantic segmentation models via auxiliary cells". In: *Proc. of CVPR*. 2019, pp. 9126–9135.

[81] Valeria Nikolaenko et al. "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records". In: *Proc. of IEEE S&P*. 2013.

[82] Seyed Yahya Nikouei et al. "Real-time human detection as an edge service enabled by a lightweight cnn". In: *2018 IEEE International Conference on Edge Computing (EDGE)*. IEEE. 2018, pp. 125–129.

[83] Nvidia. *Megatron-LM*. https://github.com/NVIDIA/Megatron-LM. Online; accessed in April 2024.

[84] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[85] Hyungjun Oh et al. "ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference". In: *CoRR* abs/2404.07947 (2024). Accepted to ASPLOS 2024 (summer cycle). URL: https://arxiv.org/abs/2404.07947.

[86] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[87] Jan Peters, Katharina Mülling, and Yasemin Altün. "Relative Entropy Policy Search". In: *Proc. of AAAI*. 2010.

[88] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *Proc. of ICLR*. 2018.

[89] Ramya Prabhu et al. "vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention". In: *ArXiv* abs/2405.04437 (2024). URL: https://api.semanticscholar.org/CorpusID:269614548.

[90] William H Press and Saul A Teukolsky. "Kolmogorov-Smirnov Test for Two-Dimensional Data: How to tell whether a set of (x, y) data paints are consistent with a particular probability distribution, or with another data set". In: *Computers in Physics* 2.4 (1988), pp. 74–77.

[91] Jennifer S Raj and S Jennifer. "Optimized mobile edge computing framework for IoT based medical sensor network nodes". In: *Journal of Ubiquitous Computing and Communication Technologies (UCCT)* 3.01 (2021), pp. 33–42.

[92] Pranav Rajpurkar et al. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264. URL: https://aclanthology.org/D16-1264.

[93] Francisco Romero et al. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021.

[94] Muhamad Risqi U Saputra et al. "Distilling knowledge from a deep pose regressor network". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 263–272.

[95] Seldon. *Deploying Large Language Models in Production: LLM Deployment Challenges*. https://www.seldon.io/deploying-large-language-models-in-production-llm-deployment-challenges. Accessed: 2024-11-08. 2023.

[96] T. Sen and H. Shen. "Machine Learning based Timeliness-Guaranteed and Energy-Efficient Task Assignment in Edge Computing Systems". In: *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. 2019.

[97] Tanmoy Sen and Haiying Shen. "A Data and Model Parallelism based Distributed Deep Learning System in a Network of Edge Devices". In: *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*. 2023.

[98] Tanmoy Sen and Haiying Shen. "Distributed Training for Deep Learning Models On An Edge Computing Network Using Shielded Reinforcement Learning". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*.

[99] Wonik Seo et al. "SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms". In: 18.4 (July 2021).

[100] ShareGPT. *sharegpt-english*. https://huggingface.co/datasets/theblackcat102/sharegpt-english. 2023.

[101] Divyasheel Sharma and Santonu Sarkar. "Enabling Inference and Training of Deep Learning Models for AI Applications on IoT Edge Devices". In: *Artificial Intelligence-based Internet of Things Systems*. Springer, 2022, pp. 267–283.

[102] Ragini Sharma et al. "Are existing knowledge transfer techniques effective for deep learning with edge devices?" In: *2018 IEEE International Conference on Edge Computing (EDGE)*. IEEE. 2018, pp. 42–49.

[103] Ying Sheng et al. "Fairness in Serving Large Language Models". In: *Proceedings of OSDI*. 2024.

[104] *Source code is available at GitHub link:* /r/Elastically-Scheduling-Multiple-DNN-Inference-Jobs-on-an-Edge-Device-for-High-Efficiency-and-Accurac-B2FD/.

[105] Allan Stisen et al. "Smart Devices Are Different: Assessing and MitigatingMobile Sensing Heterogeneities for Activity Recognition". In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 2015.

[106] Biao Sun et al. "Llumnix: Dynamic Scheduling for Large Language Model Serving". In: *Proc. of OSDI* (2024).

[107] Tianxiang Tan and Guohong Cao. "Efficient Execution of Deep Neural Networks on Mobile Devices with NPU". In: *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)*. 2021.

[108] Zeyi Tao and Qun Li. "eSGD: Communication Efficient Distributed Deep Learning on the Edge". In: *Proc. of HotEdge*. 2018.

[109] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

[110] *tcconfig*. https://github.com/thombashi/tcconfig.

[111] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices". In: *Proc. of ICDCS*. 2017.

[112] *TensorFlow Model Benchmark Tool Description*. https://https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/benchmark. Accessed: 2020-4-27.

[113] *The Qualcomm Snapdragon Graphics Pipeline Explained*. https://www.penguinsolutions.com/about-us/newsroom/qualcomm-snapdragon-graphics-pipeline.

[114] *Top 4 Pre-Trained Models for Image Classification with Python Code*. https://tinyurl.com/533dfdkc.

[115] S. De Vito et al. "On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario". In: *Sensors and Actuators B: Chemical* 129.2 (2008).

[116] Cheng-Cheng Wan et al. "ALERT: Accurate Anytime Learning for Energy and Timeliness". In: *Proc. of Usenix ATC* (2020).

[117] Jiayu Wang et al. "Poster: Maintaining Training Efficiency and Accuracy for Edge-assisted Online Federated Learning with ABS". In: *Proc. of ICNP*. 2020.

[118] S. Wang et al. "When Edge Meets Learning: Adaptive Control for Resource-Constrained Distributed Machine Learning". In: *Proc. of INFOCOM*. 2018.

[119] Shiqiang Wang et al. "When Edge Meets Learning: Adaptive Control for Resource-Constrained Distributed Machine Learning". In: *Proc. of INFOCOM*. 2018.

[120] *wondershaper*. https://github.com/magnific0/wondershaper.

[121] A. Wong, Z. Lin, and B. Chwyl. "AttoNets: Compact and Efficient Deep Neural Networks for the Edge via Human-Machine Collaborative Design". In: *Proc. of CVPRW*. 2019.

[122] Bingyang Wu et al. "dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 911–927. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/wu-bingyang.

[123] Bingyang Wu et al. "Fast Distributed Inference Serving for Large Language Models". In: *ArXiv* abs/2305.05920 (2023). URL: https://api.semanticscholar.org/CorpusID:258588170.

[124] Jing Wu et al. "HiTDL: High-Throughput Deep Learning Inference at the Hybrid Mobile Edge". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4499–4514. DOI: 10.1109/TPDS.2022.3195664.

[125] Mengwei Xu et al. "Approximate query service on autonomous iot cameras". In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 2020, pp. 191–205.

[126] Ran Xu et al. "ApproxDet: Content and Contention-Aware Approximate Object Detection for Mobiles". In: *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 2020.

[127] XX. *FLEX*. https://anonymous.4open.science/r/Elastically-Scheduling-Multiple-DNN-Inference-Jobs-on-an-Edge-Device-for-High-Efficiency-and-Accurac-B2FD/README.md. 2023.

[128] Bo Yang et al. "Edge intelligence for autonomous driving in 6G wireless system: Design challenges and solutions". In: *IEEE Wireless Communications* 28.2 (2021), pp. 40–47.

[129] Haichuan Yang, Yuhao Zhu, and Ji Liu. "End-to-End Learning of Energy-Constrained Deep Neural Networks". In: *Proc. of ICLR*. 2019.

[130] Mao Yang, Yafei Dai, and Xiaoming Li. "Bring Reputation System to Social Network in the Maze P2P File-Sharing System". In: *International Symposium on Collaborative Technologies and Systems (CTS'06)*. 2006.

[131] Qian Yang et al. "Ouroboros: On Accelerating Training of Transformer-Based Language Models". In: *Proc. of NeurIPS* (2019).

[132] Shuochao Yao et al. "DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework". In: *Proc. of SenSys*. 2017.

[133] Shuochao Yao et al. "FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices". In: *Proc. of Sensys*. 2018.

[134] Ashkan Yousefpour et al. "ResiliNet: Failure-Resilient Inference in Distributed Neural Networks". In: *Proc. of FL-ICML*. 2020.

[135] Gyeong-In Yu et al. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022.

[136] Chengliang Zhang et al. "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving". In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. 2019.

[137] Jiajun Zhang et al. "A Locally Distributed Mobile Computing Framework for DNN based Android Applications". In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. ACM. 2018, p. 17.

[138] Susan Zhang et al. "OPT: Open Pre-trained Transformer Language Models". In: *ArXiv* (2022).

[139] Wenyu Zhang et al. "MASM: A Multiple-algorithm Service Model for Energy-delay Optimization in Edge Artificial Intelligence". In: *IEEE Transactions on Industrial Informatics* (2019).

[140] Xiangyu Zhang et al. "Shufflenet: An extremely efficient convolutional neural network for mobile devices". In: *Proc. of CVPR*. 2018.

[141] Yu Zhang, Tao Gu, and Xi Zhang. "MDLdroidLite: A Release-and-Inhibit Control Approach to Resource-Efficient Deep Neural Networks on Mobile Devices". In: *Proc. of MobiSys*. 2020.

[142] Yihao Zhao et al. "Multi-Resource Interleaving for Deep Learning Training". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022.

[143] Youpeng Zhao, Di Wu, and Jun Wang. "ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)* (2024), pp. 1005–1017. URL: https://api.semanticscholar.org/CorpusID:268691374.

[144] Zhihe Zhao et al. "Ecrt: An edge computing system for real-time image-based object tracking". In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2018, pp. 394–395.

[145] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018).

[146] Zangwei Zheng et al. "Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=eW233GDOpm.

[147] Yinmin Zhong et al. "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 193–210. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[148] H. Zhou, Soroush Bateni, and Cong Liu. "S^{3}DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2018), pp. 190–201.

[149] Li Zhou, Hao Wen, et al. "Distributing deep neural networks with containerized partitions at the edge". In: *Proc. of HotEdge*. 2019.

[150] Hang Zhu et al. "Network planning with deep reinforcement learning". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021.

[151] Qianchao Zhu et al. "SampleAttention: Near-Lossless Acceleration of Long Context LLM Inference with Adaptive Structured Sparse Attention". In: *ArXiv* abs/2406.15486 (2024). URL: https://api.semanticscholar.org/CorpusID:270702543.

[152] Awrahman Zmnako et al. *Run machine learning inference workloads on AWS Graviton-based instances with Amazon SageMaker*. AWS Machine Learning Blog. https://aws.amazon.com/blogs/machine-learning/run-machine-learning-inference-workloads-on-aws-graviton-based-instances-with-amazon-sagemaker/. 2022.

[153] Katarina Zvarikova, Jakub Horak, and Peter Bradley. "Machine and Deep Learning Algorithms, Computer Vision Technologies, and Internet of Thingsbased Healthcare Monitoring Systems in COVID-19 Prevention, Testing, Detection, and Treatment". In: *American Journal of Medical Research* 9.1 (2022), pp. 145–160.