DATA ALLOCATION AND BENCHMARKING IN PARALLELIZED MOBILE EDGE LEARNING

By

DUNCAN J. MAYS

A thesis submitted to the School of Computing in conformity with the requirements for the Degree of Master of Science

Queen's University

Kingston, Ontario, Canada

January, 2023

Abstract

Democratizing Edge Computing (EC) by tapping into the copious yet underutilized computational resources of IoT devices can facilitate the use of Mobile Edge Learning (MEL). In MEL, it is important to address system heterogeneity in a way that minimizes staleness to improve learning accuracy, particularly in Parallelized Learning (PL). To do so, a centralized data allocation approach is typically used. However, this approach tends to overlook the privacy of learners, since learners' capabilities are assumed to be known beforehand by the orchestrator. In this context, we propose the Data Allocation via Benchmarking (DAB) scheme. DAB is a decentralized data allocation scheme that eliminates staleness and achieves a certain Quality of Service (QoS), while preserving the privacy of learners. DAB also introduces a novel method to enable each learner to accurately estimate its own hardware characteristics via benchmarking. In addition, we propose the Minimize Expected Delay (MED) scheme. MED enables multi-task allocation for PL under uncertainty in learners' capabilities. Given the state probabilities of learners, MED makes uncertainty-aware decisions by formulating the data allocation problem as an Integer Linear Program (ILP) that aims to minimize the sum of the maximum expected delay of all tasks, while abiding by certain training time and budget constraints. Furthermore, we propose a novel MEL framework, called Axon, to foster testing on real testbeds. Extensive performance evaluations on a real testing environment show that DAB outperforms a prominent representative of the centralized data allocation approach by up to 12% and 26% in terms of loss and prediction accuracy, respectively. In addition, the proposed benchmarking scheme yields an 83% reduction in benchmarking error compared to a prominent baseline scheme. Performance evaluations also show that MED outperforms an uncertainty naive baseline by up to 10% and 42% in terms of training time and data drop rate, respectively.

Co-Authorship

Journal Articles

- D. J. Mays, S. A. Elsayed and H. S. Hassanein, ”Uncertainty-Aware Data Allocation for Parallelized Mobile Edge Learning,” 2022, (Journal paper in preparation)

Conference Publications

- D. J. Mays, S. A. Elsayed and H. S. Hassanein, ”Decentralized Data Allocation via Local Benchmarking for Parallelized Mobile Edge Learning,” 18th International Wireless Communication and Mobile Computing Conference (IWCMC), Dubrovnik, Croatia, 2022.

Acknowledgment

I would like to thank Prof. Hossam Hassenein for his guidance and support. I am especially grateful for Sara Elsayed, for without her patience and support this thesis would likely not have been written. I also thank my girlfriend Jacqueline, our two cats, and any friend I met here who made Kingston my home more that the city I go to school in.

I'd like to give honorable mentions to Byan Hoang, Edan Parker, Jack Malloch, Micheal Treagus and Jared McGrath for using axon in their fourth-year capstone project. My thanks to you for the interest in the project and for being willing to test it out on your capstone!

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DAB	Data Allocation via Benchmarking
DUB	Data Upper Bound
CPU	Central Processing Unit
CSA	Centralized Staleness Aware
EEC	Extreme Edge Computing
EED	Extreme Edge Device
FL	Federated Learning
FLOPS	FLoating Point Operations Per Second
GPU	Graphics Processing Unit
GUC	Global Update Cycle
GUI	Global Update Index
HA	Heterogeneity Aware
HTTP	HyperText Transfer Protocol
HU	Heterogeneity Unaware
IDL	Interface Definition Language
ILP	Integer Linear Programming
IoT	Internet of Things
LAN	Local Area Network

MEC	Multi-access Edge Computing
MED	Minimize Expected Delay
MEL	Mobile edge Learning
ML	Machine Learing
MMTT	Minimize Maximum Training Time
MNIST	Modified National Institute of Standards and Technology
MTPL	Multi-Task Parallel Learning
OS	Operating System
PC	Personal Computer
PL	Parallel Learning
PPM	Predictive Performance Modelling
PyPI	Python Packag Index
QoS	Quality of Service
RAM	Random Access memory
ReLU	Rectifier Linear Unit
REST	REpresentational State Transfer
SA	Staleness-Aware
SB	Subset Benchmarking
SGD	Stochastic Gradient Descent
SU	Staleness Unaware

TCP	Transfer Control Protocol
URL	Universal Resource Locator
WiFi	Wireless Fidelity
WSGI	Web Server Gateway Interface

Chapter 1

Introduction

1.1 Overview and Motivation

With the proliferation of the Internet of Things (IoT), it is expected that by 2027, 41 billion IoT devices will come online, generating an additional 800 zettabytes of data [1] [2]. The time-sensitive nature of this data is expected to force 90% of analytics to be performed at the edge to avoid latency of transmission to remote data centers, which is highly incurred in cloud computing due to the need to transmit excessive amount of data to remote data centers [3].

Multi-access Edge Computing (MEC) has emanated as a promising computing paradigm that can significantly curtail latency by providing the computing service within proximity of end-users [4]. However, the preponderance of existing MEC models and platforms are contingent on infrastructure-based edge nodes that are entirely managed by cloud service providers and/or network operators [5]. This oligopoly can be evaded by exploiting the prolific yet underutilized computational resources of IoT devices, also referred to as Extreme Edge Devices (EEDs), such as smartphones, PCs, autonomous vehicles, etc. Parallel processing at EEDs can help democratize the edge and authorize more players to construct and govern their own edge cloud [6]. Democratizing the edge can create a new edge computing tech market that is people-owned, democratically governed, and accessible/profitable to all. Moreover, parallel processing at EEDs enables the computing service to be provided much closer to end-users. This can dramatically reduce the delay, which can help foster a wide range of data-intensive and latency-sensitive applications that require intensive processing at the edge (i.e., edge processing) [7]. Training Machine Learning (ML) models at the edge is one example of edge processing [8].

Performing ML in a distributed manner, which is referred to as Distributed Learning (DL), has gained significant momentum lately [9]. In particular, Mobile Edge Learning (MEL), which enables ML models to be collaboratively trained on a collection of resource-constrained EEDs, has been capturing the attention of the research community [6]. This can be attributed to the ongoing and substantial increase in the number of EEDs (i.e., learners). Despite being resource-constrained individually, the collective power of such devices can be significantly profuse. The integration of these abundant yet underutilized computational resources with MEL provides a promising edge learning paradigm for a broad range of IoT applications, such as object detection, speech recognition, and virtual and augmented reality [9] [10].

MEL can be categorized into two categories; Parallelized Learning (PL) and Federated Learning (FL) [9] [10]. In PL, a global orchestrator transmits randomly picked subsets of data to each learner, whereas in FL, the learners train on locally stored datasets [9]. PL is used when the orchestrator lacks computational resources to train an effective model, and thus chooses to distribute the workload to a set of distributed learners [10]. In FL, the learners collect their own data and can take advantage of models trained on a larger dataset while keeping their own data private [10]. Both PL and FL involve a set of distributed learners, where each learner trains a model independently and in parallel, and then the orchestrator performs an aggregation process, by which these models are turned into a single, more generalizable model [11] [12]. Determining the amount of data that needs to be allocated to each learner (i.e., data allocation) is one of the most crucial decisions in PL.

1.2 Challenges

Making efficient data allocation decisions for PL at the extreme edge can be hindered by the following challenging issues:

1. **Device Heterogeneity:** PL at EEDs entails the usage of user-owned devices, such as smartphones, connected vehicles, tablets, etc. Such devices have varying compute and computation capabilities, as they possess different hardware, power needs, and network connection quality. In order to utilize such heterogeneous devices, any workload distribution system must have a way to determine the capabilities of these devices, whether through prior knowledge of hardware specifications, benchmarking, or usage monitoring [13] [14]. Failing to address device heterogeneity can limit PL to the performance of its weakest learner [10] [15]. Most existing benchmarking schemes in ML strive to characterize learners based on the number of floating point operations that they can execute every second [16]. However, they fail to accurately estimate the rate at which each learner can train a given model.

2. **Staleness:** Staleness is a metric that can be applied to two parameter updates from different learners, and that is meant to give some indication of the redundant information provided by both updates [10]. In PL, it is important to address system heterogeneity in a way that minimizes staleness to improve learning accuracy. To resolve this issue, research efforts have contemplated changing the size of the learners' local models, allocating different amounts of data to each learner, and allowing learners to iterate over their local data a variable number of iterations [10] [11]. To do so, a centralized data allocation approach is typically used. However, this approach tends to overlook the privacy of learners, since learners' capabilities are assumed to be known beforehand by the orchestrator. Thus, there is a need for data allocation schemes that minimize/eliminate staleness without compromising the privacy of learners.

3. **Uncertainty in Device Capabilities:** Since EEDs are user-owned devices, they are

subject to a dynamic user access behavior. In particular, at any given time during training, users can access their devices to stream a video, or play a video game, or run any intensive application/learning task, which can affect the computation and communication capabilities of learners. This dynamic user access behavior and resource contention can trigger a high level of uncertainty in the training capabilities of affected learners, which can drastically increase the training time. Such uncertainty imposes a challenging data allocation issue that has been mostly overlooked in the literature.

4. **Time criticality:** For many reasons, synchronous PL is preferred over asynchronous PL [17]. In the latter, aggregation happens intermittently after a bounded number of updates have been uploaded. In contrast, synchronous PL requires all learners to submit their parameter update before the orchestrator can aggregate the updates and continue training. Thus, it is desirable to bound the time for learners to perform their updates and avoid what is known as the Straggler Problem [18], where a small number of learners cause large delays. In addition, minimizing the training time to maintain a certain Quality of Service (QoS) is crucial in many applications and learning tasks [19].

5. **Multi-task Allocation and Recruitment Budget:** In extreme edge computing, PL is fostered by the service provider (i.e., the orchestrator), which acts as a mediator between learners and the task requester [6] [19]. The task requester submits its learning task to the orchestrator, and the latter recruits different learners to perform the learning task. To encourage participation in the service, the orchestrator has to solicit the computational resources of learners by providing them with incentives. Scenarios where the orchestrator receives multiple learning tasks and is bounded by a certain recruitment budget can be challenging in PL. This is since tasks need to contend for the resources of learners while abiding by the specified budget limit.

1.3 Objectives and Contributions

Our objectives can be summarised as follows:

1. Accurately estimating the computation and communication capabilities of heterogeneous devices to determine the time needed by each learner to train a given model on a given number of data samples.
2. Mitigating the privacy concerns associated with the learners' capabilities.
3. Eliminating staleness while abiding by a certain time limit.
4. Fostering multi-task data allocation that enables multiple tasks to contend for the available learners' resources, while maintaining certain time and budget constraints.
5. Making uncertainty-aware data allocation decisions that account for the dynamic user access behavior and minimize the training time under uncertainty.
6. Developing a simple and easy to access MEL framework that enables conducting rigorous, scrupulous, and replicable testing of developed solutions on real testbeds.

The contributions of this thesis can be summarized as follows:

1. *Subset Benchmarking (SB):* SB addresses Objective 1 and Challenge 1 by enabling each learner to locally estimate its own computational capability. In contrast to most existing benchmarking schemes in ML that characterize learners based on the number of floating point operations that they can execute every second [20] [16], SB strives to accurately estimate the rate at which each learner can train a given model. To estimate the compute capability of devices, each learner trains a dummy model on training data while measuring the rate at which gradient updates are performed. This dummy model has a matching architecture to the network being trained, but the parameters are kept separate. SB exploits the iterative nature of backpropagation, where training a neural

network is essentially repeating the same process for each batch in an epoch, and for every epoch until convergence. Thus, we benchmark learners by running a subset of the learning task on the learner, and extrapolating the runtime to the whole workload. This is in contrast to past methods which attempt to model the computation of the training process. SB can also be used to estimate the download bandwidth of learners by the same means. A small number of samples is downloaded to each learner, and the time to download a larger number of samples is calculated using the download rate in samples per second.

2. *Data Allocation via Benchmarking (DAB):* DAB addresses Objectives 2 and 3 as well as Challenges 1, 2, and 4. It strives to eliminate staleness while preserving the privacy of learners by considering the learners' hardware characteristics as private data that cannot be shared with the orchestrator. To do so, DAB introduces a decentralized data allocation approach, where each learner determines the upper bound on the amount of data that it can process such that a certain training time threshold (i.e., deadline) is not exceeded. This decentralized data allocation regime, where learners choose how much data to train on rather than having it assigned by the orchestrator, not only preserves learner privacy but also allows learners to choose their level of participation. To the best of our knowledge, DAB is the first decentralized data allocation scheme in PL that eliminates staleness while preserving the learners' privacy and abiding by a certain training deadline.

3. *Minimize Expected Delay (MED):* MED addresses Objectives 4 and 5, as well as all given challenges by considering multi-task data allocation that accounts for the different states at which the learners can be, and the probabilistic impact of such states on their capabilities. MED models variations in learner performance as uncertainty in learner capabilities and allocates data for multiple tasks in the presence of such uncertainty. Given the state probabilities of each learner (i.e., learner capabilities),

MED formulates the data allocation problem as an Integer Linear Program (ILP) to minimize the expected delay of all incoming tasks while abiding by a certain budget and deadline limit. In contrast to existing schemes that fail to consider the dynamic user access behavior and its impact on the capability of learners during the training task, MED accounts for the uncertainty that stems from such a dynamic behavior and performs uncertainty-aware data allocation decisions.

4. *Axon:* To address Objective 6, we develop a Python Remote Procedure Call (RPC) framework based on HyperText Transfer Protocol (HTTP), which we refer to as Axon. In contrast to most existing MEL schemes that rely on simulation-based testing, we use Axon to conduct extensive experiments on a real testbed of heterogeneous Raspberry Pi and Jetson Nano devices. Axon facilitates function calls between multiple hosts on the network. It is an open-source framework that can be leveraged by the research community, since it is simple and easy to use compared to other prominent MEL frameworks. In addition, Axon is flexible and expressive, due to its ability to perform all algorithms and communication patterns used in MEL.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a description of different edge computing paradigms, presents a literature review of MEL, benchmarking, data allocation in PL, and MEL frameworks, followed by a discussion of Axon. Chapter 3 introduces DAB, as well as the underlying benchmarking scheme SB, and discusses the performance evaluation of DAB and SB compared to prominent data allocation and benchmarking schemes. Chapter 4 presents MED, and provides a detailed discussion of its performance evaluation compared to baseline data allocation schemes. Chapter 5 concludes the thesis and highlights some potential future research directions.

Chapter 2

Background and MEL Frameworks

2.1 Edge Computing and Mobile Edge Learning

2.1.1 Edge Computing

Recent years have seen a massive growth in the amount of compute in edge networks. The effective utilization of this compute would open the way for new services and compute-heavy applications running on distributed edge devices. The number of smart devices worldwide exceeds double the number of humans, with nearly 90% of the world's population owning such a device and 53% having mobile internet [21] [22]. This growth in the number and the capability of edge devices has been paired with an increase in demand for compute-heavy applications, such as augmented reality, video streaming, and machine learning. To service such applications, Cloud Computing (CC) refers to the transmission of application data to backend servers where the request is fulfilled and transmitted back to the client on the edge. To rely on CC for the applications of the future would be to place an undue load on network links [23]. Edge Computing (EC) has been proposed as a paradigm which processes application data on servers close to the edge of the internet, thus avoiding the need to transmit data to backend servers [24]. Extreme Edge Computing (EEC) is a paradigm which take this notion a step further and fulfills service requests by utilizing latent compute on edge devices themselves, thus moving computation as close as possible to the requesting user [25].

To keep up with demanding applications that require large amounts of compute or data, many applications depend on offloading tasks to the backend via CC. Moreover, the power constraints of mobile devices pose a major obstacle for compute-heavy mobile applications. Thus CC has arisen which refers to the use of large data centres containing highly accessible compute resources that can be configured to service a wide range of tasks [26]. Cloud providers allow developers to pay for partial tenancy of their resources, enabling developers to deploy web services without the need to acquire server hardware. These web services provide the backend for applications reliant on CC, however, this method incurs excessive latency and load on network links [23].

The key issue in cloud computing is its centralized nature. Cloud data centres can be multiple network hops away from a requesting users, and so cloud offerings have failed to capture latency-sensitive applications. Edge Computing (EC) is a computing paradigm involving the deployment of web services on infrastructure pieces close to the users of those services [27]. This is done to minimize the latency of access for the users, by removing the long-distance communication typically required by Cloud Computing alternatives. EC is not meant to replace Cloud Computing, but complement it. Compute and data intensive tasks require centralization, and so EC will handle latency sensitive applications while operating in concert with the cloud.

EC relies on dedicated infrastructure outside of the LAN of the requesting user. In contrast, Extreme Edge Computing (EEC) fulfills service requests by utilizing latent compute on edge devices themselves, thus moving computation as close as possible to the requesting user [25]. This is done both to reduce latency as much as possible, and as a way to democratise compute to ensure free access [28]. Much as before, EEC will not replace EC or CC, but will operate alongside them. The most latency sensitive applications will be processes by edge devices local the requesting user, which defer less latency sensitive tasks to servers at the edge, which themselves defer to the cloud. In IoT literature, this concept has been referred to as the Cloud-to-Things continuum [29].

2.1.2 Mobile Edge Learning

MEL can be split into two categories: Federated Learning (FL) and Parallelized Learning (PL) [9] [10]. In PL, a global orchestrator transmits randomly picked subsets of data to each learner, whereas in FL, the learners train on locally stored datasets [9]. PL is motivated by the desire to recruit computational resources to the training task. It is used when the orchestrator lacks the computational resources to train an effective model, and so chooses to offload the task to a set of distributed learners [10]. In FL, learners collect their own data which they treat as private and never transmit to another entity on the network. FL allows learners to access a model that is trained on a much larger dataset than the data they possess themselves, while also allowing them to preserve the privacy of their own data.

Both FL and PL involve a set of distributed learners, where each learner trains a model independently and in parallel with the other learners [12] [8]. Both algorithms also involve an orchestrator, which obtains parameter updates from the learners periodically, and aggregates all learners' parameters into a single, more generalizable model [12] [8]. The two schemes require learners to perform repetitive update routines, which consist of the following three steps:

1. Learners download parameters, and data as well in the case of PL
2. Learners train the downloaded parameters independently and in parallel
3. Learners upload their trained parameters to the orchestrator

After one such routine, known as a Global Update Cycle (GUC), the orchestrator aggregates the parameter updates from each learner. This aggregation is typically done through parameter averaging, weighted by the number of batches each update has been trained on by the corresponding learner [8]. Previous research efforts have shown that the loss surfaces of properly designed neural architectures are convex, and so averaging the weights and biases of partially trained networks within these architectures is statistically likely to bring the parameters closer to their optimal value [30].

A major issue in FL is data heterogeneity [18]. Data heterogeneity, which is also referred to as statistical heterogeneity, represents the differences in the distribution of the training data gathered on each device in FL. Data heterogeneity causes the local models trained by each learner to diverge in parameter space, and significantly reduces the effectiveness of federated averaging [31]. Data heterogeneity is a consequence of FL's requirement for private data. In contrast to FL, data privacy is not an issue in PL. Past research has found that the negative effects of data heterogeneity can be avoided with as little as 1% of data being intelligently shared between devices, and since all data is shared in PL, data heterogeneity is not a concern [32].

As opposed to data heterogeneity, device heterogeneity is a shared concern in both FL and PL, particularly in extreme edge computing [18]. Device heterogeneity refers to the variations in the training capabilities of the learner devices. While some FL research addresses system heterogeneity, it is a primary concern in PL [33]. This is since data allocation decisions in PL are highly dependent on the available computational resources of learners.

2.2 Data Allocation in Parallel Learning

In this section, we provide a review of the existing data allocation schemes in PL. Note that data allocation schemes can be classified based on the aggregation mode into synchronous and asynchronous schemes [18], based on heterogeneity into heterogeneity-aware (HA) and heterogeneity-unaware (HU) schemes [10], and based on staleness into staleness-aware (SA) and staleness-unaware (SU) schemes [34]. A taxonomy of such schemes is depicted in Figure 2.1.

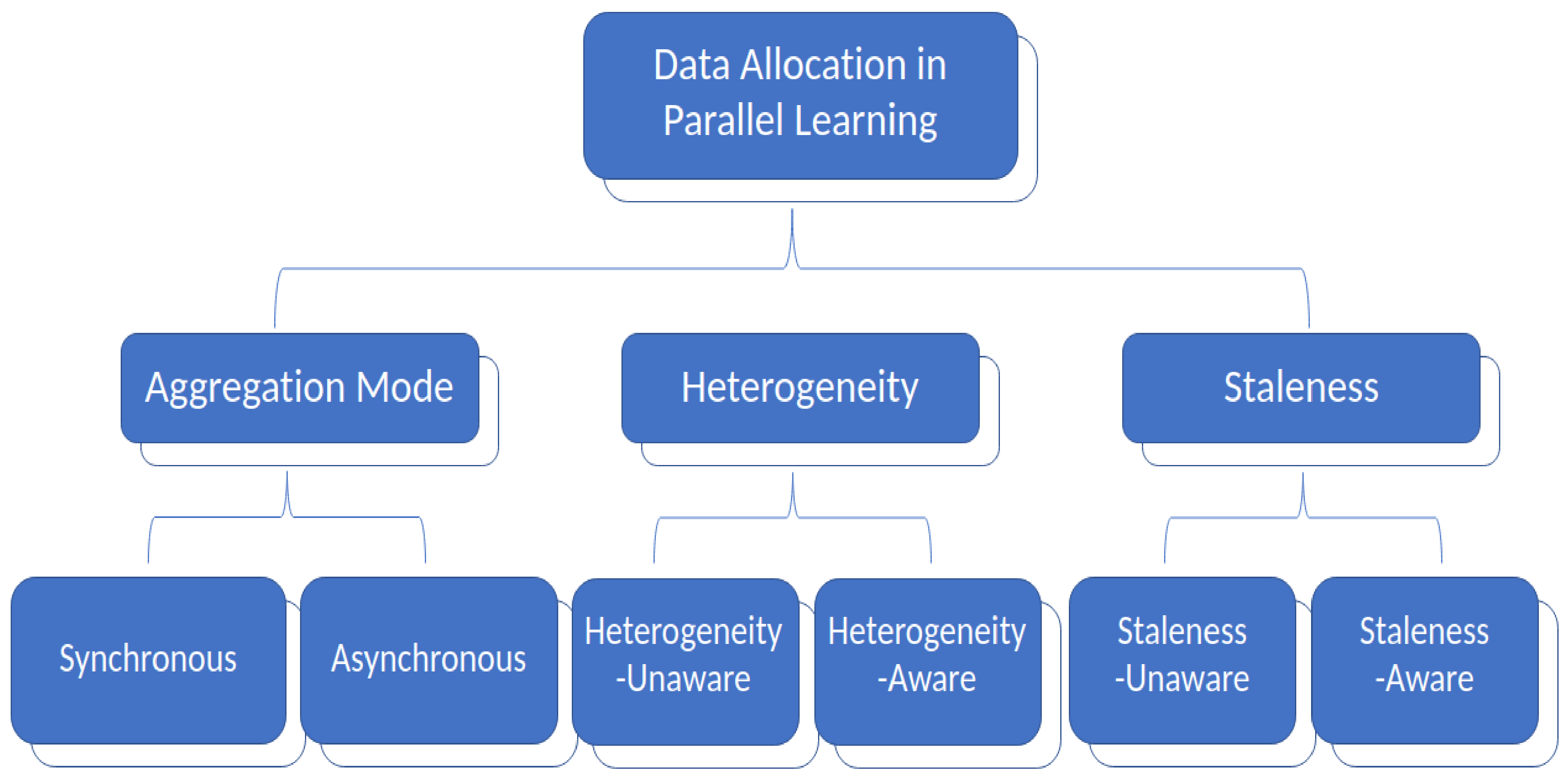


Figure 2.1: Taxonomy of data allocation in PL

2.2.1 Synchronous vs. Asynchronous Methods

An important distinction in parallel learning research, is that between asynchronous and synchronous methods [18]. In synchronous methods, all learners must complete training and upload their updates before aggregation can occur [8]. This means that the time for a single GUC is the time for the slowest learner to complete its local update routine [20]. In contrast, in asynchronous methods, aggregation happens intermittently after a bounded number of updates have been uploaded, thus avoiding the drawback of waiting for the slowest learner [34].

FedAvg [8] remains the defacto standard implementation of synchronous PL. A significant issue facing synchronous PL, which is derived from system heterogeneity, is known as the straggler problem [1] [18]. Stragglers refer to a small number of workers who take a long time completing updates. Since aggregation cannot occur until all learners have completed, stragglers can slow down training to an unacceptable degree [1] [18]. Past research in synchronous PL has mitigated the straggle problem by selecting learners which are unlikely to be stragglers, either by analysis of past update delays [35] or by benchmarking learners before training begins [33].

The simplest solution to the straggler problem however, is to aggregate once an acceptable number of updates have been submitted, thus asynchronous PL is motivated by speeding up training in the presence of stragglers and system heterogeneity in general [17] [36]. Asynchronous PL trades the problem of stragglers for that of late or stale updates [1]. These are updates that have been submitted late, after other updates have been incorporated into the central model. It has been shown that naively incorporating stale updates can be detrimental to learning accuracy [34] [37]. Thus, asynchronous PL methods need some way to weight stale updates less to mitigate these effects [36] [38]. Note that asynchronous PL is very sensitive to hyper-parameter values, such as the learning rate, and also performs poorly in the presence of data heterogeneity [1]. Broadly speaking, asynchronous PL can only approach the performance of synchronous PL in learning metrics, and its convergence still relies on assumptions that learners take a bounded amount of time to train [18]. Due to these disadvantages, the synchronous approach to PL is far more common [1] [17] [18].

In this thesis, we adopt the synchronous approach to PL due to the aforementioned reasons. To ensure that GUC can occur in a bounded time despite system heterogeneity, we allocate tasks such that the training time for each learner is bounded by a certain deadline. Learners are aware of this deadline and will halt training if they have not completed training before the deadline.

2.2.2 Heterogeneity Aware vs. Heterogeneity Unaware

System heterogeneity is a major issue in MEL [1] [18] [39]. Thus, it is useful to divide past PL research into two groups; Heterogeneity Aware (HA) algorithms and Heterogeneity Unaware (HU) algorithms, as has been done by past works [10] [40].

HA algorithms speed up training by adjusting the size of the task given to each learner to reduce training time. The training time of a single GUC in synchronous PL, is the time for the slowest worker to complete training and upload their parameters [20]. Thus HU algorithms which give the same sized task to each learner are limited in speed by their weakest learner. Here we review three categories of methods that past research in HA MEL has used to adjust the size of the task given to each learner in order to achieve this; changing the model sent to each learner for training [15], having the learners train for differing numbers of training iterations [41], and adjusting the amount of data given to each learner [33].

Previous research has changed the model being trained on each learner to adjust the computational complexity of the local update routine, proportionally to each learner's training ability. Two methods to do this are network quantization and network pruning. Network quantization is a technique where different learners will encode neural network parameters with different number of bits, saving communication and computation during training [42] [43]. This is effective at scaling training difficulty, but can result in parameter divergence and slow training speed [44]. As a result, [45] remains the only paper known to the authors to have applied the concept in a federated setting. HeteroFL [15] uses lottery ticket methods from [46] to prune the global model into multiple sub-models of different sizes. Smaller models go to weaker learners so that they can train the same number of samples per second as stronger workers with larger models. Upon aggregation, all sub-models are expanded back to full size and averaged as normal. This has the disadvantage of being computationally intense for the orchestrator, since it must prune the global model into many sub-models, and so is a poor fit for our purposes.

A simple solution to the problem of system heterogeneity is to allow learners to perform as many training iterations as they are able to in the allotted time [47] [48]. This would automatically allocate tasks of the appropriate size to each learner. However, this also introduces parameter staleness, which is a metric defined on two parameter updates as the difference in the number of training iterations performed on each of the two updates [34] [36]. Parameter staleness correlates with loss and is known to cause slow training [40].

A third way to alter the size of training task given to each learner, is to adjust the size of the dataset transmitted to each learner. This does not produce parameter staleness, and is not computationally intense for the orchestrator. A common approach is to use an optimization formulation to allocate data to minimize or maximize an objective function

[10] [33] [40]. Some more novel methods have been used to allocate data as well, such as a genetic algorithm [20], and a Stackelberg game [49]. Due to the advantages of not causing staleness and being computationally simple for the orchestrator, we chose this approach for our work. In our MEL model, the orchestrator uses an optimization formulation to allocate variable amounts of data to each learner.

While past research has addressed system heterogeneity with a variety of effective methods, most works assume the orchestrator possesses the required knowledge of learner characteristics a priori. Sparingly few works in MEL deal with the crucial task of determining learner characteristics [33]. In contrast, our work provides a benchmarking technique which can be used to determine the training capabilities of edge devices. We also present a novel scheduling regime which does not require the transmission and accumulation of this information on the orchestrator, preserving learner privacy.

2.2.3 Staleness Aware vs. Staleness Unaware

Past research in MEL has been divided by weather or not the method addresses the issue of update staleness. Parameter staleness is a metric that can be applied to two parameter updates, calculated as the difference in the number of training iterations performed on each update. Parameter staleness correlates with loss, and is known to cause slow training and poor performance of converged neural networks. Two categories of methods have been used to mitigate staleness, past works have allocated tasks to reduce staleness and have also weighted stale updates less during federated averaging.

Some works allow some staleness to better utilize powerful workers with fast network connections. The works of [48] [50] allowed learners to perform a different number of training iterations while preserving accuracy. These works are in the field of FL where data allocation is impractical. One past work in PL has jointly allocated data batches and training iterations to minimize neural loss [51]. This method considered theoretical bounds on the difference between the training loss at any given iteration and the optimal loss, from which the authors were able to derive an expression for the objective function in terms of the number of local updates.

While past authors have allowed fast learners to train further than others, despite the staleness it causes, others have weighted stale updates less to avoid the negative impact they have on learning results. This is common in asynchronous PL, where stale updates are seen as unavoidable. Two past works, [34] and [36], have presented methods to gainfully aggregate stale updates by an adaptive weighting scheme during averaging.

We opted to instead mitigate system heterogeneity by changing the amount of data allocated to each learner, rather than allowing the learners to perform different numbers of training iterations and accept some staleness. Our work can be considered HA and SU.

2.3 Benchmarking and Predictive Performance Modeling

Predictive Performance Modeling (PPM) is an important component in many distributed systems, being used to improve resource efficiency and user satisfaction [52]. PPM encompasses a family of techniques used to predict execution durations and resource usage for jobs and tasks which can then be fed into a scheduler. Past works in MEL have required runtime estimations for the training loop of neural networks on distributed learners to perform scheduling decisions [10] [33], and thus there has been recent interest in PPM for deep learning.

An early and simple method of PPM for deep learning relies on a performance metric known as FLoating point Operations Per Second, or FLOPS [14]. Since training neural networks is a mostly mathematical operation, the peak FLOPS that a GPU is capable of has been used to characterize its training ability [16]. One method for predicting training loop duration for an arbitrary neural network on a target GPU from [14] is to time the training loop on a reference GPU, then scale the training loop duration with the FLOPS ratio of the reference and target GPUs. This method was found to be inaccurate, with max and mean errors of 64.9 percent and 42.5 percent respectively [14]. The problem with this approach is that deep neural network training has a complex and esoteric relationship with hardware characteristics, a more complex model is needed [53].

Habitat is an open-sourced Python library that can predict the runtime of the training loop of neural networks defined using PyTorch on NVIDIA GPUs [14]. Like the FLOPS-based method from the same paper, Habitat works by first gathering data while training a neural architecture on a reference GPU, which it uses to predict the duration of a training loop on a target GPU. Rather than focus on a single metric, Habitat monkey-patches system calls and uses CUDA events [54] to obtain a profile of timing information for individual kernel operations. Using these profiles, Habitat then performs a sophisticated technique known as wave-scaling in combination with pre-trained multi-layer perceptrons to predict runtime of training loops on the target GPU [14]. While Habitat is very effective, it is complex and narrow in scope. Habitat only works for PyTorch modules on NVIDIA GPUs, and recreating wave-scaling for arbitrary training hardware would require many work-hours from skilled developers, which is simply not accessible to the average researcher. Habitat cannot be relied upon to make runtime predictions for the wide variety of devices found in edge networks. In contrast, our PPM method is extremely simple and can be readily implemented on any edge computing system.

Paleo is a performance model for deep neural networks that can predict execution delays for inference and training in model parallel and data parallel settings [55]. Paleo uses a linear FLOPS-based method to predict computation delays, and models communication delay as the amount of data to be transmitted divided by the network bandwidth between the relevant hosts. Their method considers network bandwidth and GPU FLOPS as inputs to their model, and makes no attempt at determining them automatically. Our method likewise models communication delay as the amount of data divided by the bandwidth, and has a similar model for computation time. In contrast to Paleo, determining the computation and communication capabilities of learners is a key part of our method.

Our method involves timing the training task directly, and then scaling the timing information to larger tasks. Since training neural networks is essentially the same backpropegation process for every batch of training data, we can scale the timing for a certain number of batches to a different number of batches. Communication capabilities are determined by timing data downloads to each learner. This requires the ability to dispense training tasks to the target device, which the orchestrator implicitly possesses with reference to each learner in order to conduct the PL algorithm.

2.4 MEL Frameworks in Python

In this section, we review the Python-based libraries that allow communication between multiple hosts on a network for the purpose of orchestrating MEL. We choose the Python programming language due to its use as the primary language of Machine Learning (ML). A recent survey has shown that 57% of data scientists use Python, and 33% of developers indicate it as their programming language preference [56]. This popularity in the ML community means that deep learning libraries in Python, such as PyTorch and Tensorflow, are extremely performant and stable. However, using Python can also restrict the communication libraries to only those that are available in the language. We provide an overview of the existing MEL frameworks and discuss the pros and cons of each of them within the problem context of MEL. In addition, we propose a custom-built Python framework, referred to as Axon, which can be used as an educational tool for distributed computing, MEL, and RPC. All frameworks discussed here, including Axon, are available for download on the Python Package Index (PyPI) through the command-line tool pip.

2.4.1 gRPC

gRPC is a cross-platform, open-sourced RPC framework that was developed and is maintained by Google. An RPC framework allows the programmer to trigger the execution of subroutines in another address space (typically a different computer across a network) using similar semantics to a normal function call. Using an RPC framework, the programmer can

Algorithm 1 : IDL defining a protocol buffer for a simple gRPC service

```
 1: syntax = "proto3";
 2:
 3: // a message type
 4: message StringMsg {
 5:       string msg = 1;
 6: }
 7:
 8: // the service interface
 9: service HelloWorld {
10:       rpc simple_hello(StringMsg) returns (StringMsg) {}
11: }
```

write almost the same code weather or not the subroutine being called executes locally or on a remote system. gRPC uses event-loop concurrency in conjunction with Python's asyncio library, which allows multiple concurrent RPC calls to be made at once using async/await syntax.

2.4.1.1 Serialization

Serialization is a ubiquitous issue in distributed computing. Serialization refers to the process of turning a data structure in memory into a representation that is independent of the original address space and can be transmitted over a network. All data must be serialized before being transmitted to another host, and deserialized upon receipt.

One of the core concepts in gRPC is that of a protocol buffer, which is a data serialization format defined by the programmer in an Interface Definition Language (IDL). Protocol buffer descriptions are stored on the server and are downloaded to the client at the beginning of an interaction, the client then compiles binding code in their chosen language which allows the client and server to communicate using a common format regardless of the language each was written in. Protocol buffers allow cross-language gRPC calls between 11 of the most common programming languages, such as Python, Go, C, C++, Java, and Node [57].

See an example of a protocol buffer definition in Algorithm 1. Line 1 defines the version of the IDL used to parse the file. Afterwards, a message type is defined (lines 4-6) and then a service interface which references this message type is given (lines 9-11).

In contrast to gRPC, Axon serializes data using the Python serialization library pickle. Due to this reason, Axon is not compatible with any other language than Python. Moreover, using pickle presents a security risk. Pickle allows Python objects to specify their own deserialization procedure, and so if a transmission that was serialized with pickle is intercepted on an unencrypted channel, the hacker is able to inject code into the message that the receiving host will execute when deserializing the message. This is an unacceptable vulnerability and removes the possibility of Axon being used in production-grade code. Axon remains a viable option for research purposes however.

2.4.1.2 Code Complexity

gRPC requires the programmer to specify many details about the program, which increases the complexity of code, the time for edits to occur, and the potential for errors to develop. See an example of a definition in Python for the HelloWorld service that merely prints a greeting with a string message from the client in Algorithm 2. The serve function on lines 19-26 is the program's entry point, which instantiates a server object using a thread pool on line 20, binds it to an instance of the HelloWorldServicerClass on line 22, then exposes this instance for distributed access on lines 24-36. Note that the server script must import some functionality on lines 5 and 6. These objects would have been compiled beforehand from the protocol buffer description shown in Algorithm 1, and encapsulate the serialization procedure. The HelloWorldServicer class defines what would be referred to as the business logic of the service in a production context. The class has only a single function, other than __init__, that prints and returns a greeting upon invocation from a client.

Algorithm 3 gives an example of a Python gRPC client invoking the HelloWorld service defined in Algorithm 2 . Note that the client must also import binding code compiled from the protocol buffer definition in Agorithm 1 on lines 4-5. Afterwards, the client instantiates a channel to the server (line 10), creates a stub object (line 13), casts a string into a StringMsg (line 16), and then calls the RPC on line 19, invoking the simple_hello function on the HelloWorld server.

Algorithm 2 : Python script defining a simple gRPC service

```
import grpc
import concurrent.futures as futures

# importing pre-compiled binding code
import hello_world_pb2
import hello_world_pb2_grpc

# defining the service class
class HelloWorldServicer(hello_world_pb2_grpc.HelloWorldServicer):

    def __init__(self):
        pass

    def simple_hello(self, param, context):
        greeting = f"Hello {param.msg}!!"
        print(greeting)
        return hello_world_pb2.StringMsg(msg=greeting)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    hello_world_pb2_grpc.add_HelloWorldServicer_to_server(HelloWorldServicer(), server)

    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()

serve()
```

The main advantage of Axon versus gRPC in the MEL context is its ease of use. gRPC requires IDL code on the server to specify an interface as well as each datatype used by the interface. The business logic for the service must then parse parameters from and cast return values into protocol buffers. Boilerplate code must be written that instantiates an executor, links it with a service instance, and exposes it on a network port. Client code must manage network connections and also cast arguments and parse return values to and from protocol buffers. These steps must be performed many times over during development, adding delay and risking bugs due to programmer error. Axon requires no such code complexity, being meant to be used as an educational tool.

Algorithm 4 gives an example of an Axon service definition. Note that other than the business logic on lines 4-7, there are only three lines which contain code. line 1 imports Axon,

Algorithm 3 : invoking a gRPC service

```
 1: import grpc
 2:
 3: # importing pre-compiled binding code
 4: import hello_world_pb2
 5: import hello_world_pb2_grpc
 6:
 7: def run():
 8:
 9:     # establishing a channel to the server
10:     with grpc.insecure_channel('localhost:50051') as channel:
11:
12:         # instantiating the stub
13:         stub = hello_world_pb2_grpc.HelloWorldStub(channel)
14:
15:         # creating input
16:         rpc_input = hello_world_pb2.StringMsg(msg='John')
17:
18:         # calling RPC
19:         greeting = stub.simple_hello(rpc_input)
20:
21: run()
```

Algorithm 4 : Python script defining an Axon service

```
1: import axon
2:
3: @axon.worker.rpc()
4: def simple_hello(name):
5:     greeting = f"Hello {name}!!"
6:     print(greeting)
7:     return greeting
8:
9: axon.worker.init()
```

line 3 decorates the simple_hello function to indicate the programmer wishes to expose it to distributed access, and line 10 opens a network port to serve RPC invocations. This is a drastic reduction from the 13 extra lines of code needed to expose a gRPC service.

Algorithm 5 gives an example of an Axon client invoking an RPC. Note that in this case the loopback address "127.0.0.1" is used on line 3 for testing purposes, but otherwise this would be the IP address of a remote server. In contrast to gRPC, Axon does not require the client to cast variables into protocol buffers or instantiate a channel to the server.

Algorithm 5 : Python script invoking an Axon RPC

```
import axon

stub = axon.client.SyncSimplexStub(worker_ip=”127.0.0.1”, rpc_name=”simple_hello”)

greeting = stub(”John”)
```

Algorithm 6 : Python script linking PySyft to PyTorch

```
import torch
import syft as sy
hook = sy.TorchHook(torch)
```

2.4.2 PySyft

PySyft is an open-sourced Python framework for secure and private deep learning, primarily developed by OpenMined. The framework places emphasis on data ownership, and provides abstractions for the secure processing of private data [58]. PySyft represents data using chains of references, through which commands propagate to trigger computations while keeping the data private from the original issuer of the command. PySyft can be used to implement FL, PL, secure multiparty computation and differential privacy with a user-friendly API.

PySyft is meant to be used in concert with the deep learning framework PyTorch. It provides privacy-preserving abstractions such as remote references, while leaving local computations and autograd mechanics to PyTorch. As shown in Algorithm 6, PySyft must be linked to PyTorch using a hook class in order to extend PyTorch's functionality.

The key privacy-preserving extension that PySyft provides to PyTorch is the ability to reference tensors on remote machines. PySyft allows the programmer to direct operations using references to distributed tensors which are performed remotely without the programmer having access to the data within the tensors. This enables FL because the orchestrator can use PySyft to direct a training routine on multiple distributed learners via references to tensors and modules on each learner, using syntax very similar a normal training routine that is performed locally.

Algorithm 7 : Python script representing a PyTorch training iteration

```
1: optimizer.zero_grad()
2: output = model(data)
3: loss = criterion(output, target)
4: loss.backward()
5: optimizer.step()
```

Algorithm 7 shows an example of a single gradient computation which is repeated until convergence in SGD. There are five objects being manipulated in Algorithm 7: *data* refers to the training input, *target* refers to the training label, *model*, refers to the neural model being trained, *criterion* the loss function, and *optimizer* the object which applies gradients. Their initialization has been intentionally omitted since each of them can represent either a local PyTorch object, or a PySyft reference to a remote object. Line 1 clears the gradient attribute on each parameter, line 2 calculates the model's output on the training input, line 3 calculates the loss, and lines 4 and 5 backpropegate that loss to each parameter. PySyft takes care that the interface of a reference is identical to the interface of the remote object it refers to, which allows the references to be code compatible with local objects.

Innovations such as the code compatibility between PySyft references and PyTorch objects greatly reduce the programming difficulty of implementing MEL algorithms with PySyft. Programmers can debug training routines using local objects on their own machines before running those routines on distributed hosts. Moreover, this means programmers whom are already familiar with PyTorch can use PySyft seamlessly, without having to learn a new API.

PySyft is a dedicated MEL framework with simple syntax and a capable community of developers. It's key abstraction, the remote reference, enables privacy preserving distributed computations on PyTorch tensors. While PySyft is an extremely effective tool for MEL, it is limited to encoding computations that can be represented with numeric tensors. In contrast, Axon is a more general purpose framework, which is primarily meant to be used to conduct MEL, but can be used to perform generic distributed computations on data other that PyTorch tensors. This includes ML training routines written with other popular ML frameworks, such as TensorFlow, with which PySyft has limited integration possibilities.

2.4.3 Ray

Ray is a Python distributed computing library built off gRPC at the University of California, Berkeley. It aims to provide a universal Application Programming Interface API for distributed computing, and provides simple but general high-level abstractions which let the programmer focus on relevant details while the system manages low-level settings [59].

Ray contrasts with other works reviewed here in the way application code is distributed. For other communication protocols like gRPC, the server defines the code which is run to service client requests. On Ray, the clients define application code, which is serialized using the Python library cloudpickle and sent to servers for execution. This allows much faster development. The client does not have to manually move code onto the server and restart it, nor wait for server maintainers to do so. They merely need to edit code locally and the changes are sent to the servers automatically once the application is scheduled.

Servers log into Ray clusters they wish to provide resources to, and fulfill tasks sent to them by the cluster scheduler. Ray is designed so that clients may express computations in terms of resources instead of individual machines. A Ray application may request more resources from the cluster much like a program may request more memory to be allocated from a normal operating system. This allows Ray applications to transparently scale, but removes it as a viable option for our purposes.

Our work aims to characterize the resources available on devices, and then allocate data between them. The specific device that a task is being run on is an important concern. Ray abstracts these details from the programmer, a client is not explicitly aware of, nor can they designate, the machine that a task runs on.

One can think of APIs on a spectrum from high-level and declarative to low-level and imperative. Frequently, libraries must choose between being flexible but verbose with slow development, and being easy to use with brief syntax but being unable to represent a wide range of algorithms. gRPC is too low-level for our purposes, to use in in MEL would be to write an unnecessary amount of code specifying things like serialization details and managing network channels. Ray creates the opposite problem by hiding too many details from the programmer. Our research demands access to device-level scheduling, and since Ray does not expose such details to the programmer, it is unsuitable for our research.

Axon strikes a desirable balance between flexibility and ease of use. By focusing on generic programming concepts, such as the function, Axon can make design decisions for the programmer while not constraining their actions at a high level. Axon is verbose enough to express most algorithms in MEL, while not forcing the programmer to design extensive code alongside the main logic they desire to implement.

2.4.4 Axon

Axon is an RPC framework based on HTTP, which has been under development since 2021 to facilitate experiments for this thesis. Axon aims to be an easy-to-use educational tool that gives inexperienced programmers the ability to implement MEL algorithms on a set of devices connected over a LAN. Handling multiple concurrent processes in a distributed system can significantly increase programming difficulty, and so Axon provides the programmer with two concurrency models. This allows the programmer to choose the one that best suits their experience and needs. Axon also provides simple configuration options to control the communication patterns involved in RPC invocation.

Concurrency is an essential programming technique for distributed computing. Programming concurrency refers to the execution of multiple independent processes simultaneously, and can pose a significant challenge for the inexperienced programmer. The term concurrency model refers to the set of abstractions with which the programmer handles a set of concurrent processes. There are two popular concurrency models: event-loop concurrency and thread-based concurrency.

Event-loop concurrency is a model which is natively used in many programming lan-

Algorithm 8 : Python script invoking an Axon RPC using await syntax

```
import axon
import asyncio

stub = axon.client.CoroSimplexStub(worker_ip=”127.0.0.1”, rpc_name=”simple_hello”)

async def main():
    await stub(”John”)

asyncio.run(main())
```

Algorithm 9 : Python script invoking an Axon RPC using join syntax

```
import axon

stub = axon.client.AsyncSimplexStub(worker_ip=”127.0.0.1”, rpc_name=”simple_hello”)

handle = stub(”John”)

result = handle.join()
```

guages such as JavaScript and Go. Python supports event-loop concurrency via the asyncio library. Event-loop concurrency involves multiple uwer-level threads which submit events to a shared list. The execution environment then loops over this event list, and executes callback functions corresponding to each event. Some events are disposed of once their callback is executed, while some remain on the list and repeatedly execute a callback every time the list is iterated. Callback functions may submit events at will. Event-loop concurrency is considered cooperatively scheduled since callbacks voluntarily yield control back to the event loop. In Python's asyncio, event-loop concurrency is represented using async/await syntax. See Algorithm 8 for an example of an Axon RPC invocation using the event-loop concurrency model.

Thread-based concurrency is the oldest form of concurrency and is supported by most operating systems today. Python supports thread-based concurrency through the threading module. Thread-based concurrency relies on kernel mechanics where multiple execution stacks which share an address space can be run simultaneously on the same physical CPU. Thread-based concurrency is called preemptively scheduled since the execution of a thread can be preemptively halted by the operating system to allow another thread to execute. See

Table 2.1: MEL frameworks and their features

	Ease of Use	Device-Level Scheduling	Multiple Concurrency Models
gRPC	no	yes	no
PySyft	yes	yes	no
Ray	yes	no	no
Axon	yes	yes	yes

Algorithm 8 for an example of an Axon RPC invocation using thread-based concurrency. Note that code placed between the lines 5 and 7 will execute while the server processes the RPC, and the script will halt execution until the response from the server on line 7.

HTTP is a request/response protocol where a TCP channel is created on the request and is maintained until the response. To keep this channel open for long periods of time is costly and undesirable, and so frequently browsers and frameworks place time limits on HTTP requests, which the response must meet or the TCP channel will be closed. This is an issue for MEL since frequently ML training loops can take a long time. Axon surmounts this issue by allowing the programmer to switch into a duplex communication pattern using a simple configuration option. Duplex communication begins with the client sending an HTTP request to the server to invoke an RPC. The server immediately responds to the request with a confirmation of receipt, closes the TCP channel and begins executing the RPC. Once the invoked subroutine has completed the return value is sent back to the client with a separate HTTP request from the server. The code for the client to invoke a duplex RPC is identical to that of a simplex one, which is the default behavior of using only a single HTTP request to invoke the RPC and transmit the return value back to the client.

We now compare and contrast Axon to the other MEL frameworks reviewed here. We make our comparison using three features: ease of use, representing the simplicity of the framework's API, device-level scheduling, which refers to weather or not the client can delegate tasks to a specific device, and multiple concurrency models, which represents if the client supports thread-based and event-loop concurrency. The frameworks and their relation to each feature is given in Table 2.1.

Axon is a homegrown RPC framework that is meant to be an easy-to-use educational tool. In contrast with other MEL frameworks, Axon is the only one to support more than one concurrency model, which allows inexperienced programmers to more easily implement MEL algorithms. While it would be technically possible, to implement MEL with gRPC would be to write an unnecessary amount of code specifying things like protocol buffers. Other frameworks such as Ray and PySyft are code-concise and easy to use but do so at the cost of flexibility. Both Ray and PySyft impose limitations on the programmer that make them unsuitable for our research. Ray lacks device-level scheduling and PySyft can only distribute computations that can be encoded as numeric tensors. Axon strikes a balance between having simple syntax while also being adaptable and versatile enough to meet our needs.

In the aim to become an educational tool and facilitate other research, Axon has already been a success. In 2021 a group of four undergraduate engineering students at Queen's University used Axon to implement a parallel learning algorithm for their capstone project. The use of Axon allowed these undergraduates, who had never before worked in distributed learning, to implement and run PL experiments on a local cluster of devices over the course of two semesters.

Chapter 3

Data Allocation via Benchmarking (DAB)

In this chapter, we present the Data Allocation via Benchmarking (DAB) scheme. In DAB, we vary the amount of data allocated to each learner while fixing the number of iterations in order to eliminate staleness. In contrast to existing schemes, we propose a decentralized data allocation policy where learners' privacy is paramount. Learners' privacy is preserved by keeping any information pertaining to the learner's characteristics local to the learner, rather than transmitting it to the orchestrator. In particular, learners only communicate with the orchestrator to download data and return their trained parameters. In addition, compute characteristics are obtained via a new benchmarking scheme, referred to as Subset Benchmarking (SB). As opposed to benchmarking schemes in ML that strive to characterize learners based on the number of floating point operations that they can execute every second [16], SB strives to accurately estimate the rate at which each learner can train a given model. In contrast to most existing schemes, performance evaluation is conducted on a real testbed rather than being simulation-based.

3.1 System Model

Consider a set of n learners that are recruited by the orchestrator in exchange of some incentives, denoted $W = \{w_1, w_2, ..., w_n\}$. Upon recruitment, each learner $w_k \in W$ independently runs two benchmarks, namely c_k and b_k. The benchmark c_k represents the compute power

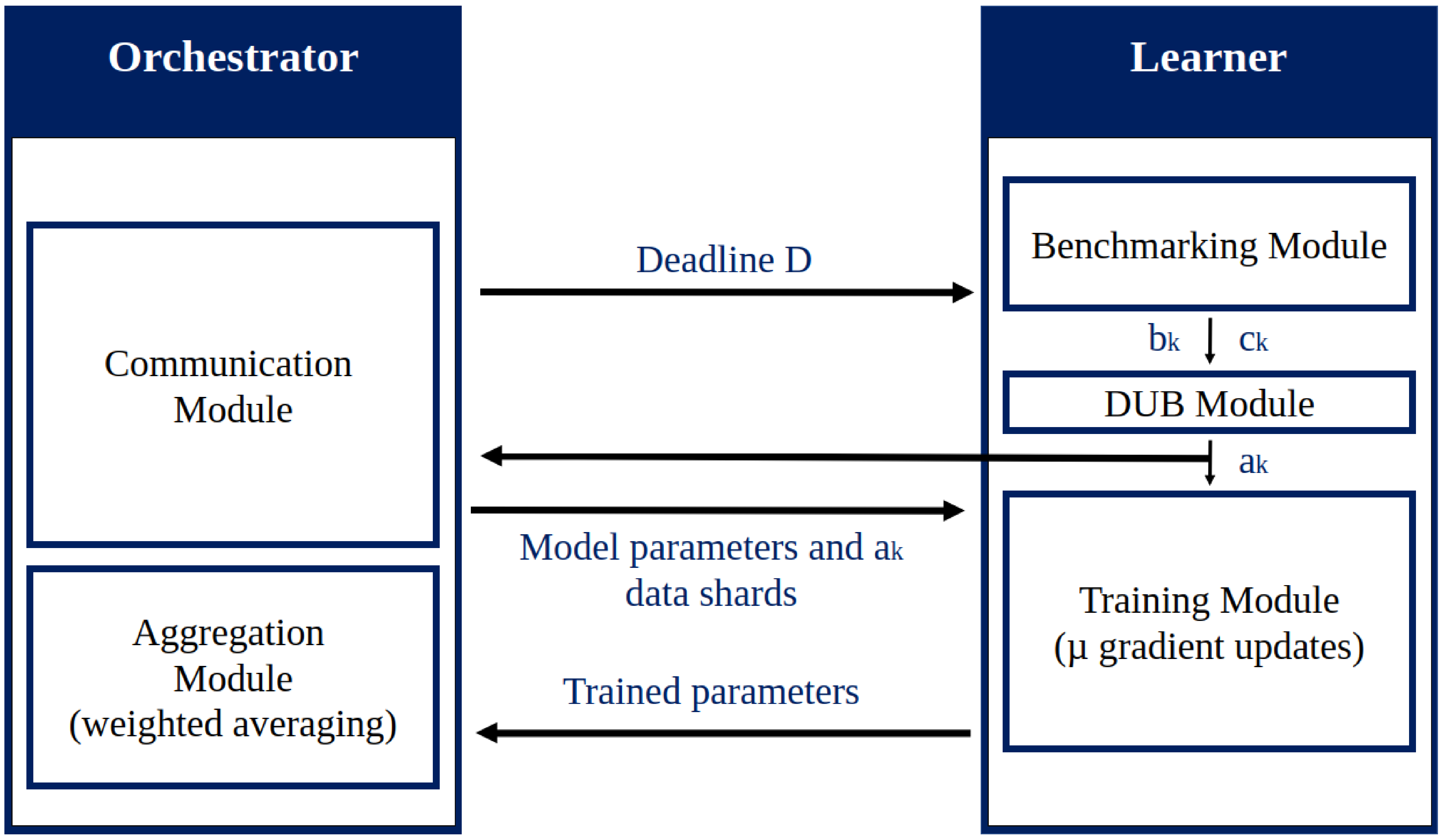

Orchestrator
Communication Module
Aggregation Module (weighted averaging)
Deadline D
Model parameters and ak data shards
Trained parameters
Learner
Benchmarking Module
bk
ck
DUB Module
ak
Training Module (µ gradient updates)

Figure 3.1: Global update cycle of DAB.

of learner w_k, which is the rate at which w_k can train the model in data samples per second. The benchmark b_k represents the download bandwidth of learner w_k, which is the bandwidth of the network connection that the learner has with the orchestrator in bytes per second. Given c_k and b_k, the learners then calculate the maximum number of data shards a_k that they can download and train for μ iterations before a given deadline D is reached. Similarly with other works [11], we assume that the channel is perfectly reciprocal within one global cycle. As depicted in Figure 3.1, the global update cycle of DAB operates as follows:

1. The orchestrator sends the training deadline D to each learner.
2. Each learner w_k runs benchmarks to obtain c_k and b_k.
3. Each learner w_k determines the upper bound a_k on the amount of data that it can download based on the estimated values of c_k and b_k.
4. Each learner w_k downloads a_k data shards, as well as the model parameters from the orchestrator.

Algorithm 10 : DAB at Learners

```
1: **Input:**
2: Number of data shards used in the benchmark $n$
3: Dummy model used in the benchmark $dummy\_model$
4: Training deadline $D$
5: Global update index $G$
6:
7: $data\_allocation(dummy\_model, n, D)$
8: **Begin**
9:   $t_d^s \leftarrow curent\_time$   // $t_d^s$ is the download start time
10:  download n & record $t_d^e$   // $t_d^e$ is the download end time
11:  $T_d = t_d^e - t_d^s$   // $T_d$ is the download time
12:  $b_k = n/T_d$   // $b_k$ of learner $w_k$, $\forall w_k \in W$
13:  $t_t^s \leftarrow curent\_time$   // $t_t^s$ is the training start time
14:  train $dummy\_model$ on n data shards & record $t_t^e$
15:  $T_t = t_t^e - t_t^s$   // $T_t$ is the training_time
16:  $c_k = n/T_t$   // $c_k$ of learner $w_k$, $\forall w_k \in W$
17:  $a_k$ is obtained from Eq. 3.5
18:  if $a_k \leq 0$
19:       return // if learner cannot train on any data, halt
20:  $numb\_shards = a_k$
21:  **for all** $g \in G$ **do**   // each iteration in G
22:      download $numb\_shards$
23:      download the most recent set of parameters from the orchestrator $\rho$
24:      **for all** $i \in \mu$ **do**   // each iteration in $\mu$
25:          train the client model on the downloaded data using $\rho$
26:          send parameters back to the orchestrator to be aggregated
27:      return training parameters
28: **End**
```

5. Each learner w_k executes μ gradient updates on its local parameters.

6. Each learner w_k uploads its parameters to the orchestrator.

7. The orchestrator averages all parameters, and steps 4-7 are repeated until convergence.

The time that a learner w_k takes to download the data and the model parameters, performs learning updates, and uploads the parameters is represented by η_k. The time for the orchestrator to perform an aggregation is negligible. For any learner w_k, η_k is bounded above by the global deadline D.

3.2 Subset Benchmarking (SB)

As demonstrated in Algorithm 10, to estimate the communication capability of devices in SB, each learner w_k downloads a predetermined number, n, of randomly selected data shards. Given the amount of time it takes to download the data shards, the learner can then infer its download speed b_k (lines 9-12). To estimate the compute capability of devices, each learner w_k trains a dummy model on this training data while measuring the rate at which gradient updates are performed to estimate c_k (lines 13-16). It is important that the model used in the benchmark has the same neural architecture as the model the orchestrator wishes to train. The weights and biases of the dummy model should be kept separate from the parameters of the model under training. Note that the dummy task represents a subset of the training task.

SB exploits the iterative nature of backpropagation, where training a neural network is essentially repeating the same process for each batch in an epoch, and for every epoch until convergence. Thus, we benchmark learners by running a subset of the learning task on the learner, and extrapolating the runtime to the whole workload.

3.3 Data Allocation Procedure

Once a learner w_k estimates its two benchmarks, namely c_k and b_k, it calculates the upper bound on the amount of data it can train within the deadline D. To determine this bound, each learner w_k first estimates the amount of time it takes to download the model parameters and the training data, which is denoted α_k and is given by Eq. 3.1, where Ω_m represents the size of the model parameters in bytes, and Ω_d represents the size of each data shard in bytes. We do not model latency in communication.

$$\alpha_k = \frac{\Omega_m + \Omega_d a_k}{b_k} \tag{3.1}$$

Each learner w_k then estimates the amount of time it takes to train the model on a_k data samples for μ iterations, which is denoted β_k, and is given by Eq. 3.2.

$$\beta_k = \frac{\mu a_k}{c_k} \tag{3.2}$$

The time that each learner w_k takes to upload its data, denoted γ_k, is then estimated as given by Eq. 3.3.

$$\gamma_k = \frac{\Omega_m}{b_k} \tag{3.3}$$

The total training time that a learner w_k takes is the sum of the time it takes to download the model parameters and the training data, train the model on a_k data samples for μ iterations, and upload its data. This total training time is denoted as η_k, and is the sum of α_k, β_k, and γ_k, as given by Eq. 3.4

$$\eta_k = \alpha_k + \beta_k + \gamma_k = \frac{\Omega_m + \Omega_d a_k}{b_k} + \frac{\mu a_k}{c_k} + \frac{\Omega_m}{b_k} \tag{3.4}$$

Each learner is required to finish μ training iterations before a certain training deadline D. Thus, the total training time η_k that each learner w_k takes must be less than D (i.e., $\eta_k < D$). As shown in Algorithm 10, given their benchmarking scores b_k and c_k, each learner calculates the maximal a_k that enables it to finish μ training iterations before D (line 17). Such a value for a_k is obtained as given by Eq. 3.5.

$$a_k < \frac{D - \frac{2\Omega_m}{b_k}}{\frac{\mu}{c_k} + \frac{\Omega_d}{b_k}} \tag{3.5}$$

Once a_k is determined, each learner w_k repeats the following steps for G number of global iterations (line 18):

1. w_k downloads a_k data shards, as well as the most recent model parameters from the orchestrator (lines 19 & 20)

2. w_k executes μ gradient updates on its local parameters (lines 21 & 22)

3. w_k sends its parameters to the orchestrator (line 23), which aggregates the received parameters and sends the average values back to all the learners.

3.4 Performance Evaluation

In this section, we evaluate the performance of DAB compared to a representative of staleness-aware centralized data allocation schemes, which is referred to as the Centralized Staleness-Aware Data Allocation (CSA) scheme [11]. We also evaluate the performance of SB compared to the baseline FLOPS benchmarking scheme [10]. We use the following performance metrics: 1) the average benchmarking runtime error, which is calculated as the average of the ratio of the difference between the predicted runtime and the actual runtime rendered by each learner to train the network, to its actual runtime, 2) the benchmarking network error, which is calculated as the average of the ratio of the difference between the predicted time to download a given number of data shards, and the measured time to download that same number of data shards, 3) the prediction accuracy, which is calculated as the ratio of correct predictions to the total number of predictions, 4) the loss of the trained network, which is calculated as the categorical cross entropy (Eq. 3.6), where y is the one-hot label and $\hat{y}$ is the output logits of the neural network, N is the dimensionality of the output, 5) training time, which is the amount of time to complete one GUC, and 6) the truancy, which is the amount of time for a GUC, minus the deadline, divided by the deadline.

$$-\sum_{i=1}^{i=N} y_i log(\hat{y}_i) \tag{3.6}$$

3.4.1 Experimental Setup

We implement DAB, CSA, SB, and FLOPS on a real testbed composed of three Raspberry Pi devices, as well as one 2070 Super, and one Jetson Nano. The hardware characteristics of each one of these devices are presented in Table 3.1. The implementation is done using our in-house Python RPC framework that we refer to as Axon [34]. Experiments are conducted

Table 3.1: Hardware Characteristics of the Testbed Devices

Device	Clock Speed	Cores	Memory	Network
2070 Super	1605 MHz	2560	8 GB	Ethernet
Jetson Nano	920 MHz	128	2 GB	Ethernet
Levono Ideapad	1200 MHz	128	8 GB	WiFi
Pi1	1.4 GHz	4	512 MB	WiFi
Pi2	1.8 GHz	4	4 GB	WiFi
Pi3	1.5 GHz	4	4 GB	WiFi

by training a feed-forward neural network as a classifier on the MNIST dataset [60]. The training set is composed of 60,000 greyscale images of handwritten digits at 28x28 resolution, with corresponding labels numbered zero through nine and is split into data shards of 500 samples each. The network architecture, which we refer to as TwoNN, is a feed-forward network with layer widths [784, 50, 20, 10], each of which is activated using the ReLu activation function, except the last layer which uses the softmax activation function. We train the network using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.1. The number of learning iterations, μ, is set to five, whereas the training deadline, D, is set to 15 seconds. The FLOPS of a learner is calculated by multiplying two 512x512 matrices 100 times, and keeping track of the time for the whole computation.

3.4.2 Results and Analysis

In our experiments, we evaluate the performance of SB and FLOPS under varying benchmark sizes. We also evaluate the performance of DAB and CSA under varying global update index, and deadline. The results procured are presented below. Experiments are repeated 30 times, and results are presented at a confidence level of 95%. Note that the rendered confidence intervals are shown on Figure 3.6. Since the confidence intervals are negligible, they are not explicitly depicted in the other figures for clarity of presentation.

3.4.2.1 The Impact of the Benchmark Size

We evaluate SB compared to FLOPS in terms of the average runtime benchmarking

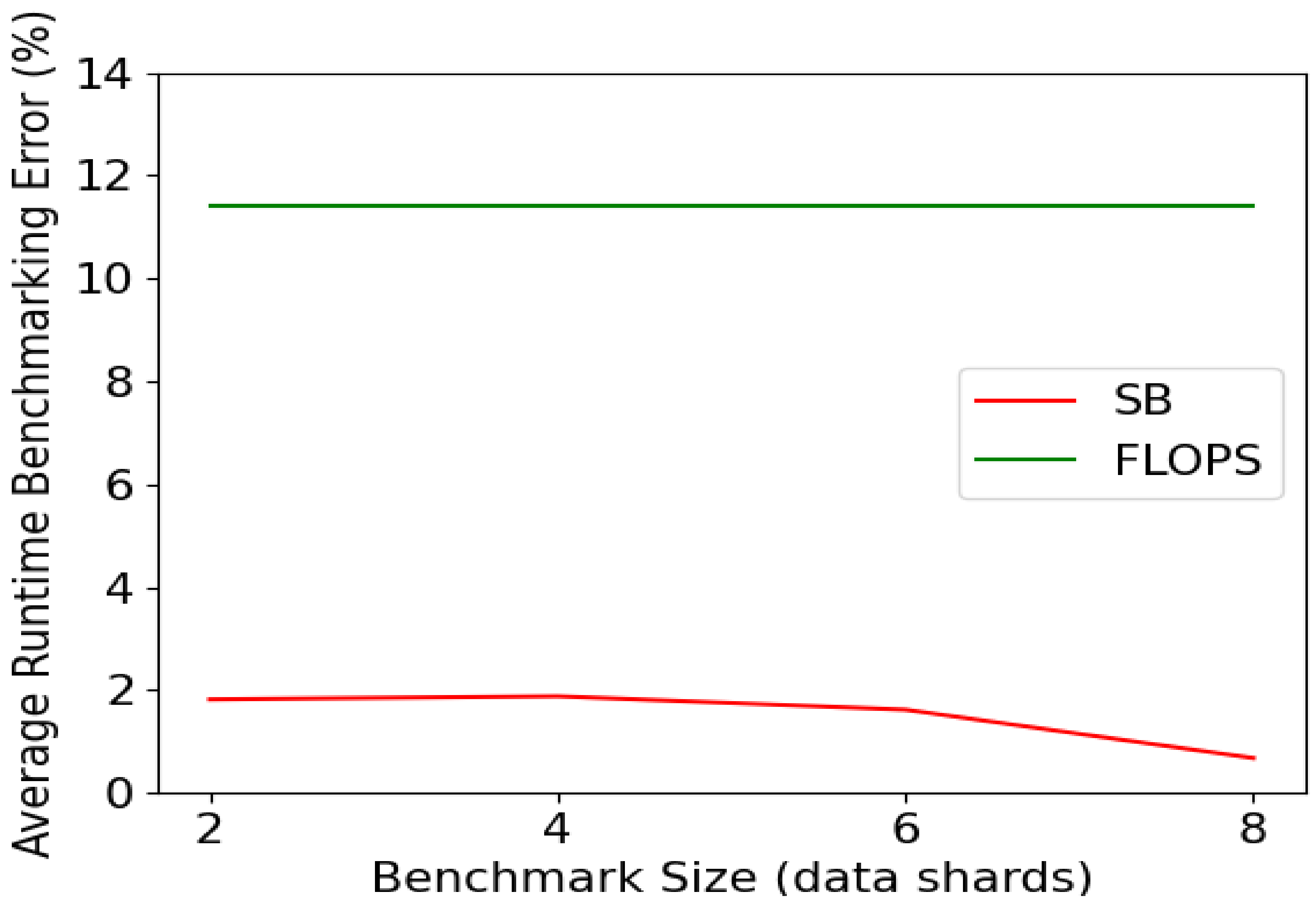


Figure 3.2: The average runtime benchmarking error of SB and FLOPS over varying benchmark size (data shards).

error over varying numbers of data shards used in the benchmark, ranging from 2 to 8. The dummy model used in this experiment is a neural network that is trained on an amount of data of 120 shards at the 2070 Super and Jetson Nano devices, and 15 data shards at each of the Raspberry Pi devices. Note that the architecture of the neural network used is the same as the one stated above. As depicted in Figure 3.2, SB yields a significant reduction in benchmarking error, reaching up to 83%. This can be attributed to the fact that FLOPS restricts its estimations of the training time to the rate at which the hardware can perform floating point operations. However, the performance of various platforms in training neural networks is highly difficult to predict, since it involves many bottlenecks other than what FLOPS is restricted to.

In contrast, SB goes beyond such a restriction by using a dummy model that mimics the actual ML model that needs to be trained, thus timing the execution of the process itself, rather than developing a model for it. Note that since FLOPS does not use data shards, it remains constant. In contrast, as the size of data shards (i.e., benchmark size) increases, the benchmarking runtime error in SB decreases. This is because the more data shards that the dummy model is trained on, the more samples the average score is made from, which

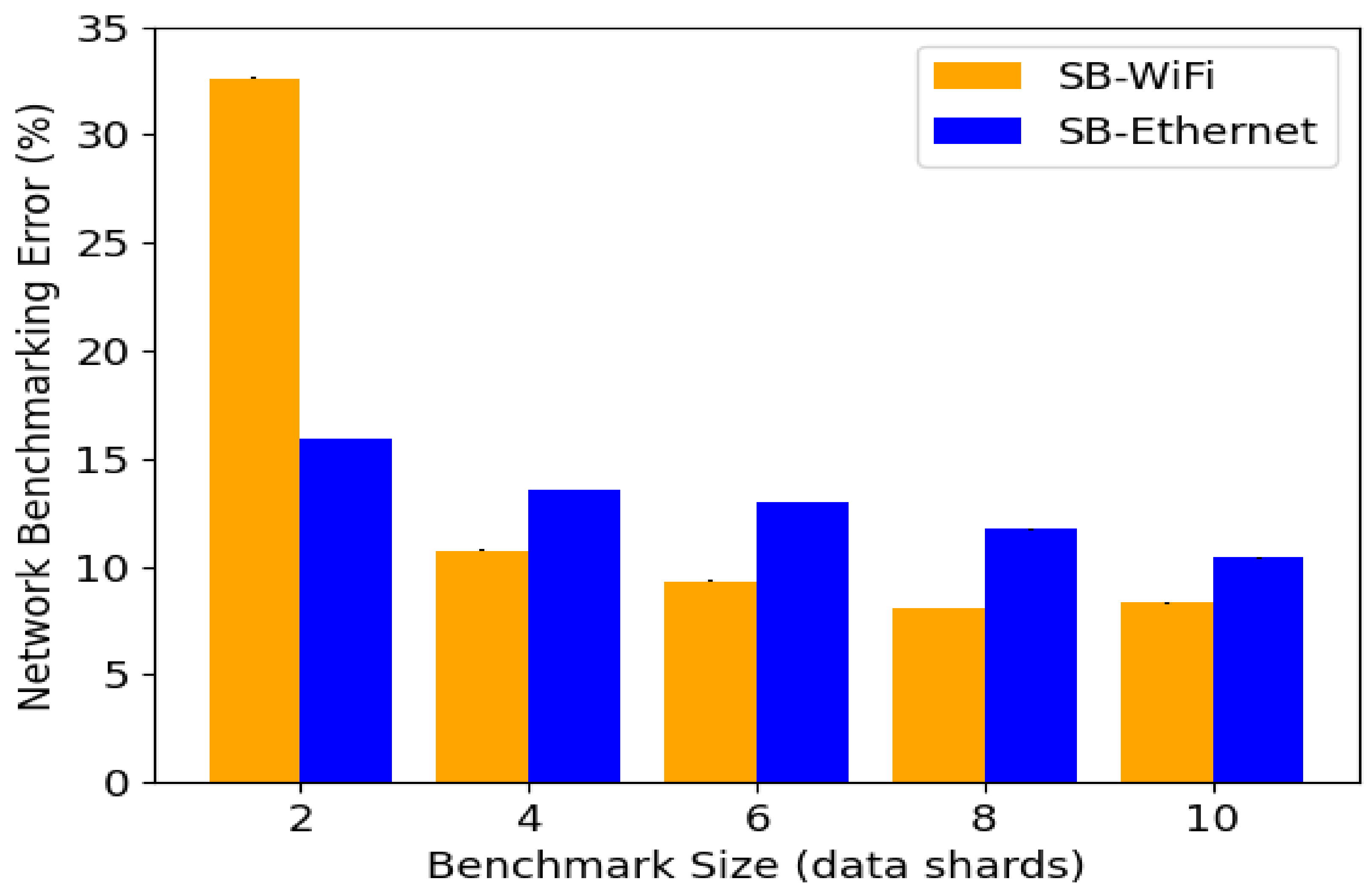


Figure 3.3: Download time prediction error for both Ethernet and WiFi over varying benchmark sizes

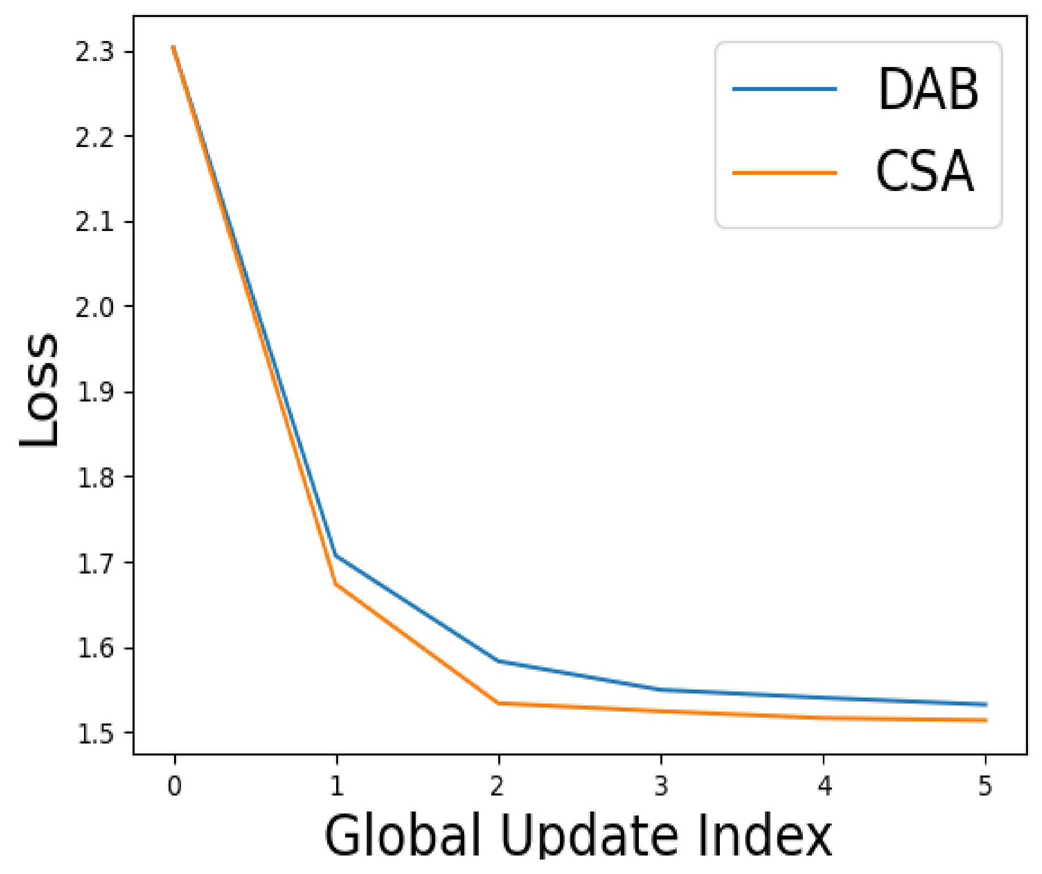
makes it closer to the actual training rate. Performance does not significantly improve when benchmark size is increased above 8 shards. It is worth mentioning that while SB has high accuracy, it is not generic. A learner's benchmarking score is specific to a single model architecture and cannot be used to predict that learner's training rate for other models.

We assess the accuracy of SB in predicting the time to download 60 shards of data over two different network connections; Ethernet and WiFi. We predict the time to perform both tasks using SB, and then report the percent error over varying benchmark sizes, ranging from 2 to 10. Each shard of data is composed of 500 samples from the MNIST dataset represented by vectors of 784 indices in length. The reported data is averaged over 45 trials on the Jetson Nano board for the Ethernet connection, and Levono Ideapad for the WiFi connection. As depicted in Figure 3.3, the error in SB prediction for the time to download 60 shards decreases as the benchmark size increases for both network types. The highest prediction error yielded on Ethernet is 16% at a benchmark size of 2 shards, and the lowest is 10% at 10 shards. On WiFi, SB has an error rate of 32% at 2 benchmarking shards, and 8% at 10 shards. Note that similar to predictions made for training time, predictions made for download time are more accurate with higher benchmark sizes, since larger benchmarks give more accurate estimations of the network bandwidth. This is due to the law of large numbers, SB effectively samples the learner's capabilities on each shard, and so more benchmarking shards bring the sample average closer to the true average. We explain that SB has a higher error on WiFi than Ethernet at small benchmark sizes, but a lower error on WiFi with large benchmark sizes, with the theory that there is latency in establishing a WiFi connection, which would skew the accuracy of our benchmark at small sizes. Generally, SB has lower error on WiFi than it does on Ethernet, with the largest improvement of 30% occurring at a benchmark size of 6 shards. This is due to the fact that Ethernet connections communicate on the same signal bus as other Ethernet connections going through the same hub, leaving them vulnerable to interference from other traffic on the network.

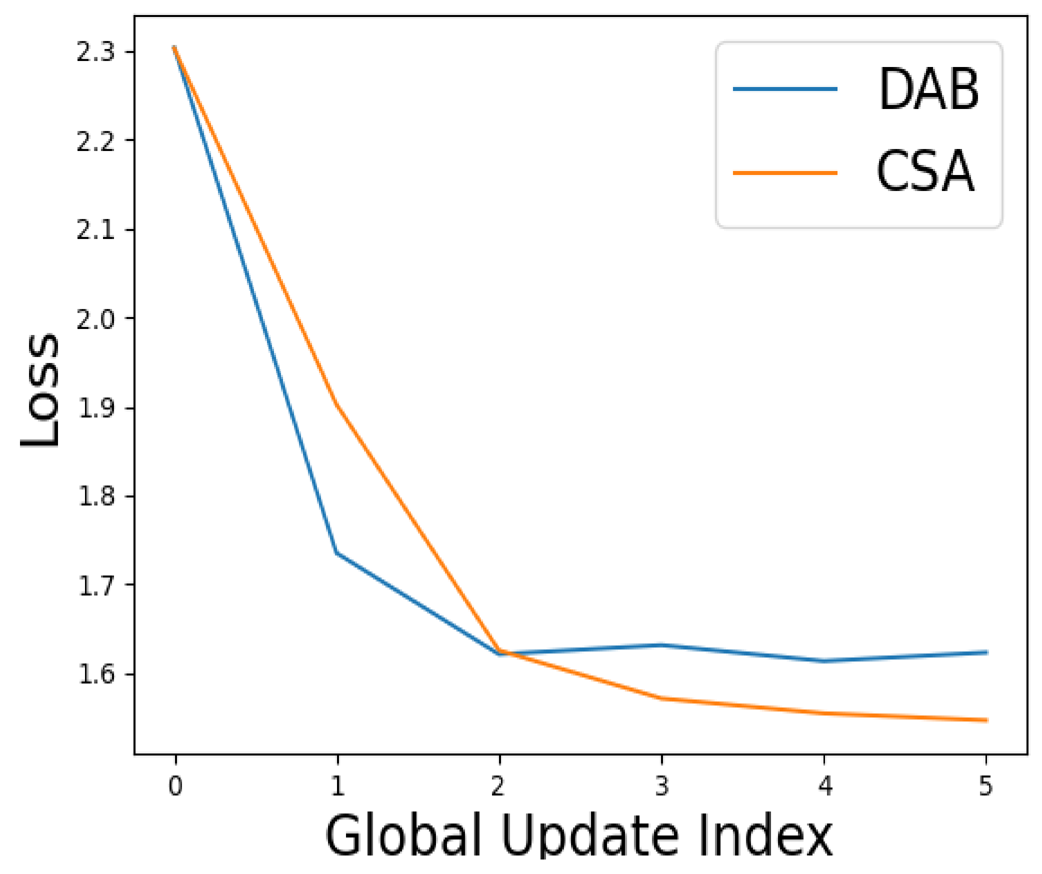
3.4.2.2 The Impact of the Global Update Index

We evaluate the performance of DAB compared to CSA in terms of loss and accuracy over varying global update index on two clusters; a cluster of two devices, Pi1 and Jetson Nano, and a cluster of four devices, Pi1, Pi2, Pi3, and Jetson Nano. At each global update, the orchestrator averages the parameters of the models from each learner, and assesses the performance of the averaged parameters.

As depicted in Figure 3.4a, the loss yielded by DAB while being trained on two devices closely approaches that yielded by CSA. This is despite being a decentralized scheme, thus lacking the global view of the hardware characteristics of the entire learners that CSA possesses. This can be attributed to the fact that DAB eliminates staleness by fixing the number of iterations, whereas CSA strives to minimize staleness under varying number of iterations. In addition, although the orchestrator does not have a global knowledge of the learners' hardware characteristics in DAB, it has a global knowledge of the upper bound of the data that each learner can download while abiding to the required training deadline. Note that DAB gets closer to CSA, with almost a 0% gap as the global update index decreases, whereas the gap starts to slightly increase by up to 6%, as the global update index increases. As the global update index increases, the leverage gained by CSA due to optimizing the number of



(a) Loss of DAB and CSA over varying global update index on a cluster of Pi1 and Jetson Nano



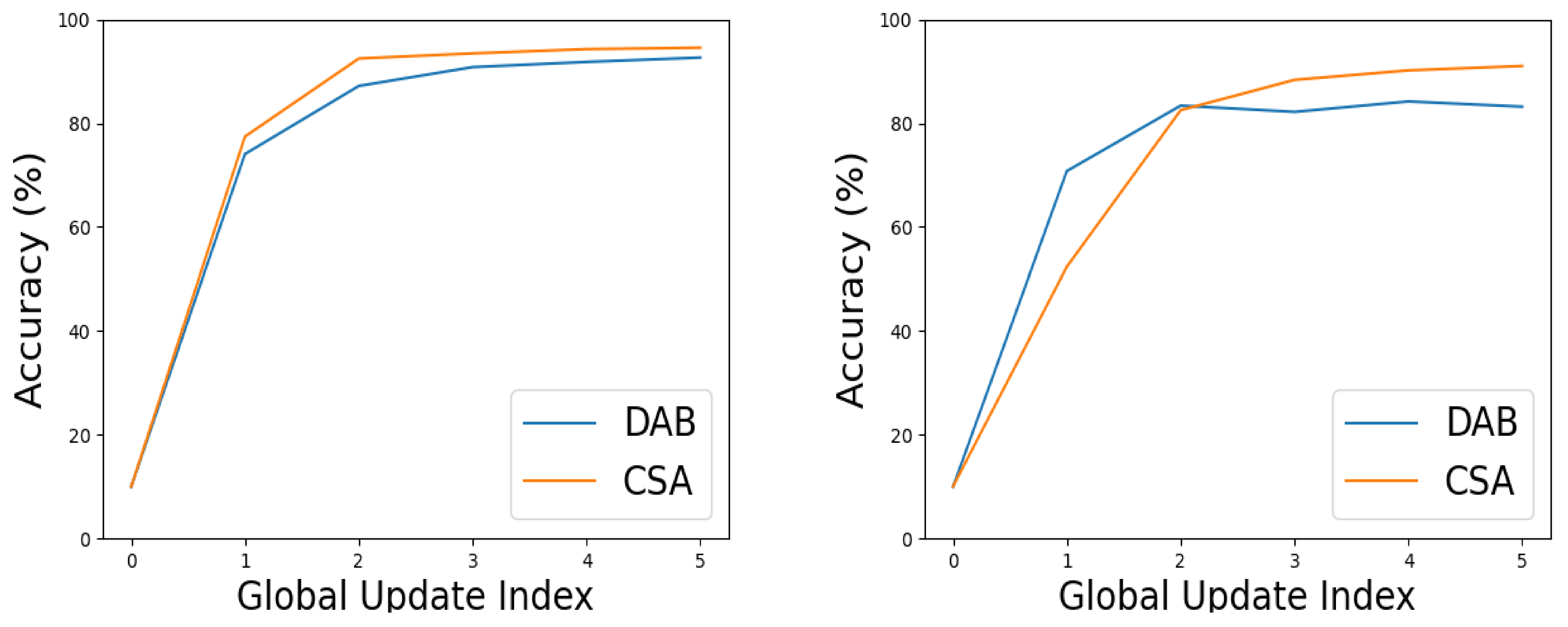
(b) Loss of DAB and CSA over varying global update index on a cluster of Pi1, Pi2, Pi3 and Jetson Nano.

Figure 3.4: Loss of DAB and CSA over varying global update index on two clusters.

learning iterations that learners perform starts to manifest.

We conduct the same experiment to assess the loss of DAB compared to CSA on a cluster of four devices. As shown in Figure 3.4b, as the global update index decreases, DAB outperforms CSA by up to 12%, whereas it starts to yield a slightly higher loss than CSA, reaching 5%, as the number of global update index increases. DAB outperforms CSA at lower global update index because CSA operates on a fixed number of data samples, which is split between learners, whereas DAB enables each learner to determine the maximum amount of data that it is able to process in the given time frame (with duplication if needed). This means that in the presence of extra training resources, DAB trains the neural network on more data per global update than CSA does, which is why it outperforms CSA when provided with four training devices but not when only two devices are available. As the global update index increases, CSA outperforms DAB, since the leverage gained by CSA due to determining the optimal number of iterations becomes more evident. Note that this requires a global knowledge that DAB does not provide for privacy reasons.

We perform the same aforementioned experiments to assess the prediction accuracy of DAB and CSA. A corresponding behavior to the loss is demonstrated in the accuracy plots



(a) Accuracy on a cluster of Pi1, and Jetson Nano. (b) Accuracy on a cluster of Pi1, Pi2, Pi3, and Jetson Nano.

Figure 3.5: Accuracy of DAB and CSA over varying global update index on two clusters.

in Figure 3.5. In the first cluster of two devices, depicted in Figure 3.5 a), DAB closely approaches CSA, with almost a 0% gap as the global update index decreases, whereas the gap starts to slightly increase by up to 2% as the global update index increases. In particular, DAB reaches a prediction accuracy of 93% compared to 95% in CSA. This is because the loss of the network trained with DAB is higher than the network trained with CSA. As depicted in Figure 3.5b in the cluster of four devices, DAB outperforms CSA by up to 26% as the global update index decreases, whereas it renders an 8% lower accuracy than CSA as the global update index increases. This can be attributed to the inverse behavior in terms of loss depicted in Figure 3.4.

3.4.2.2 The Impact of the Deadline

We assess the performance of DAB compared to CSA while varying the training deadline on a cluster of four Jetson Nanos from 21 to 30 seconds. The experiments were conducted while training a classifier with two convolution layers at 32 and 64 filters, then two feed-forward layers of 512 and 10 units. The convolutional layers are each followed by a maxpool layer that halves both dimensions of the input image. Each learner will train for one pass over their local datasets, for five global update cycles.

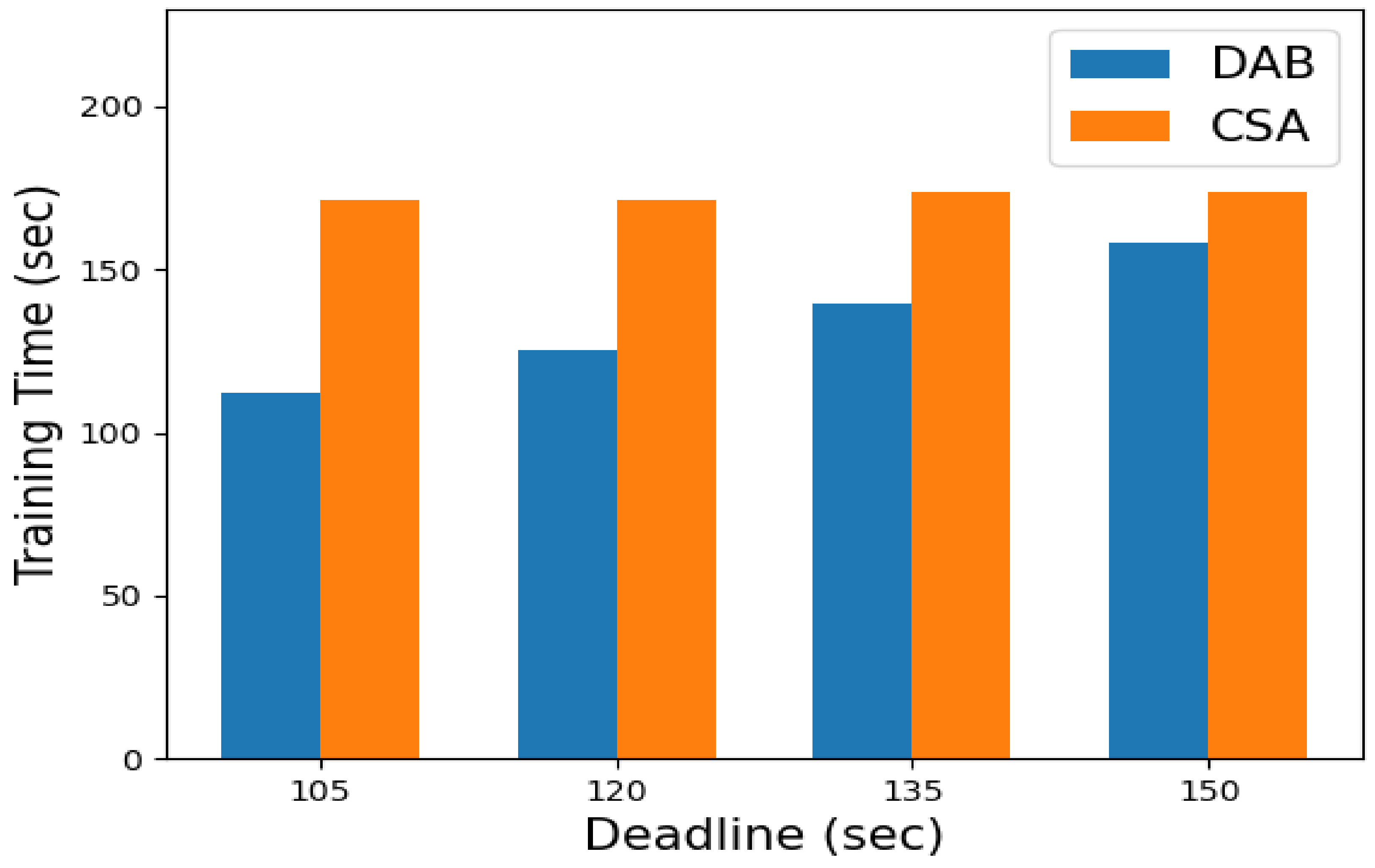


Figure 3.6: Time to complete training task for DAB and CSA on a cluster of four Jetson Nanos over varying deadline

As depicted in Figure 3.6, the training time for DAB closely follows the deadline while CSA fails to adapt and schedule tasks to meet low deadlines, despite the advantages posed by a centralized approach. Figure 3.7 gives the average truancy of each scheme, which is the amount of time the task was completed after the deadline, as a percentage of the deadline. At a deadline of 105 seconds, DAB took an average of 112 seconds to complete the training task, only exceeding the deadline by 7% while CSA took 172 seconds on average, 63% past the deadline. Despite being a centralized scheme, CSA takes at most 53% longer than DAB, 56% further past the deadline. This is due to the fact that CSA fixes the amount of data it allocates, while DAB only allocates as much data as the learners can train on in the given amount of time.

The ability of DAB to change the total amount of data allocated means DAB underperforms CSA in learning metrics by a small amount at low deadlines. Figure 3.8a shows that at a deadline of 105 seconds, the classifier trained with DAB achieves an accuracy of 91% compared to 95% for the classifier trained with CSA, a 4% reduction in accuracy. This is due to DAB reducing the amount of data it trains with to ensure the deadline is satisfied. Note that at higher deadlines, where resource contention is not as much of an issue, DAB

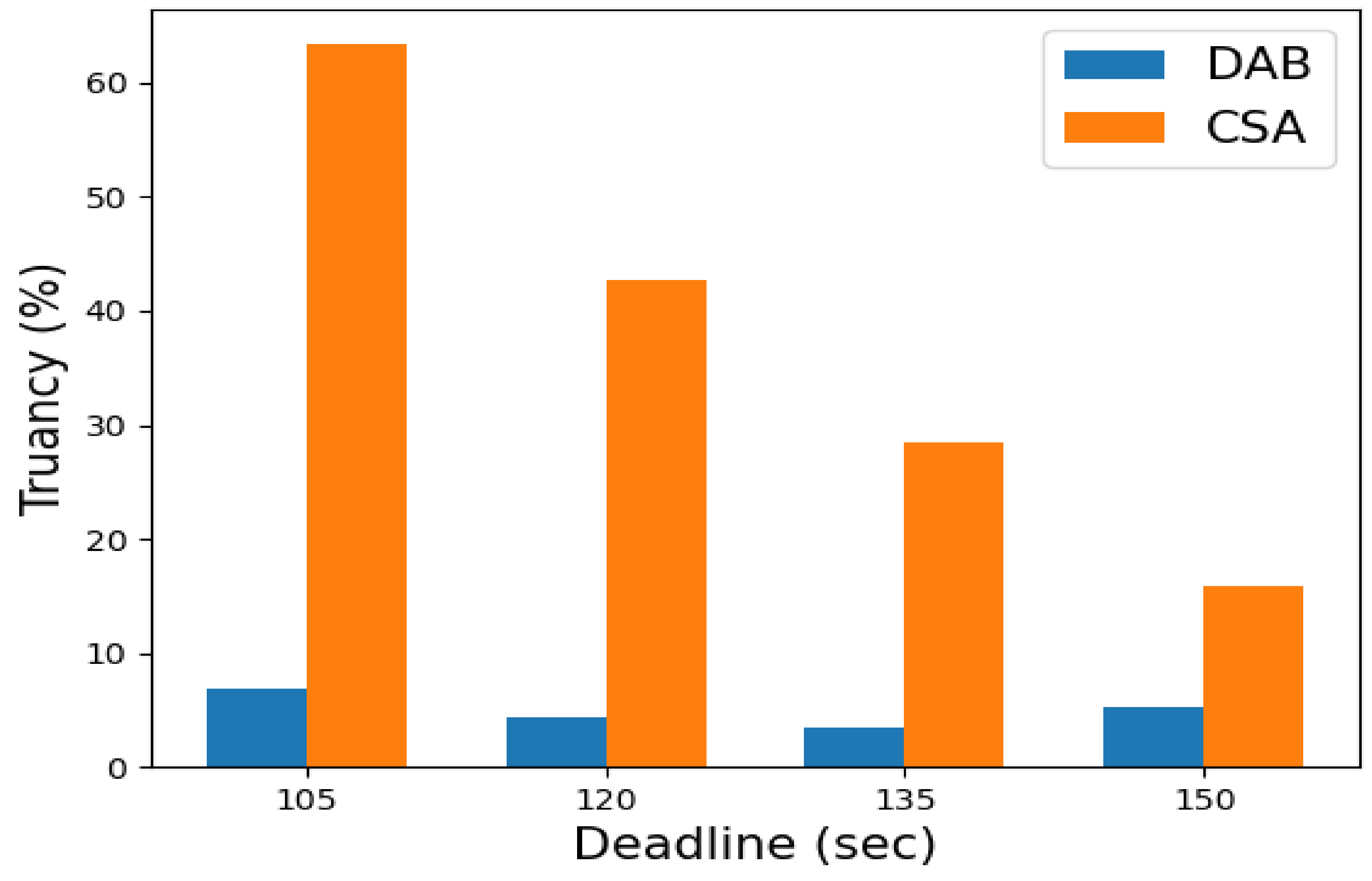
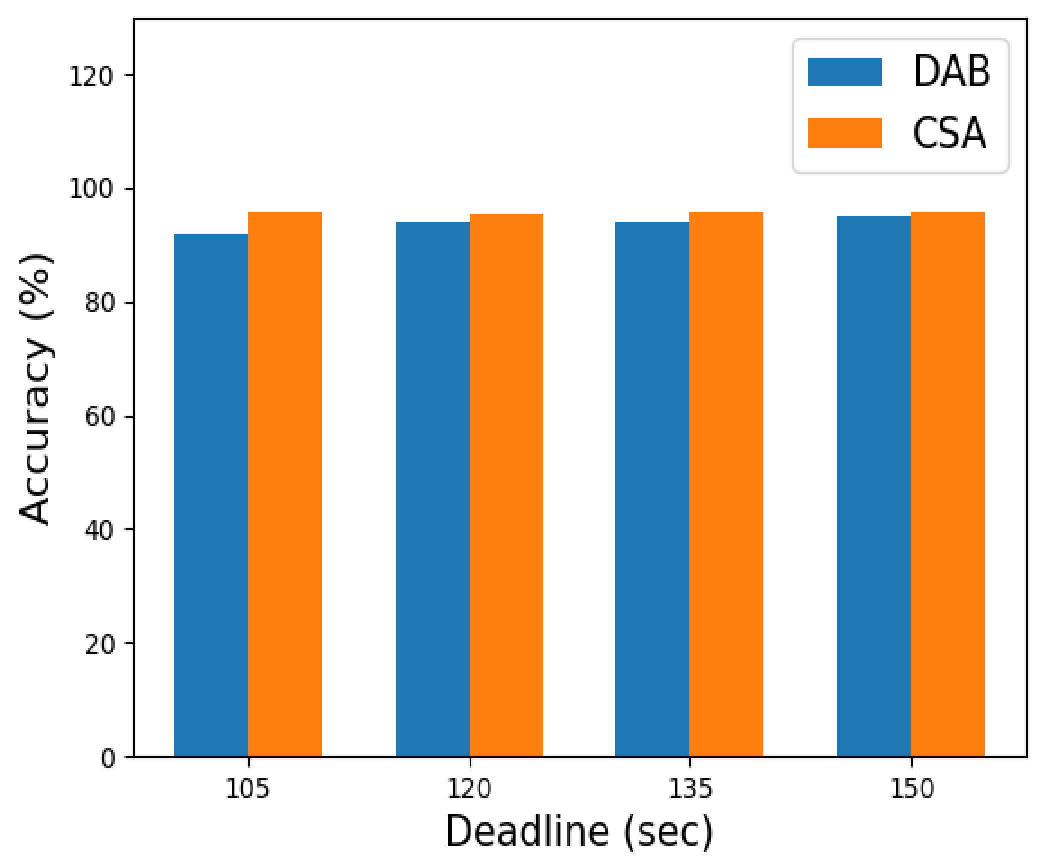
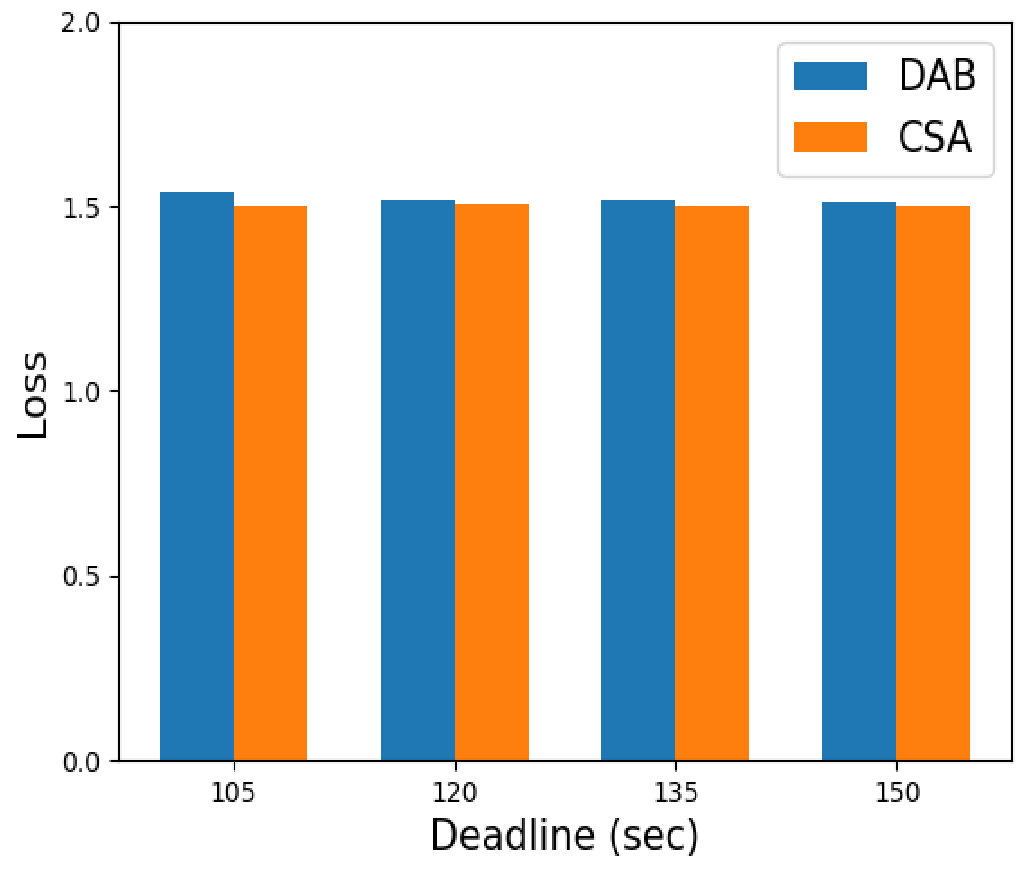


Figure 3.7: Truancy as a percentage of the deadline for DAB and CSA on a cluster of four Jetson Nanos over varying deadline



(a) Classification accuracy of neural networks trained with DAB and CSA on a cluster of four Jetson Nanos over varying deadline



(b) Neural loss of DAB and CSA on a cluster of four Jetson Nanos over varying deadline

Figure 3.8: Loss and accuracy of DAB and CSA on a cluster of four Jetson Nanos over varying deadline

approaches CSA in accuracy. At a deadline of 150 seconds, DAB has an accuracy of 95%, with CSA also being 95% accurate within a decimal of rounding error.

Corresponding results are seen in the loss plot Figure 3.8b, at a deadline of 105, DAB achieves a loss of 1.53 to 1.50 for CSA, a 0.2% difference. This matches the difference in accuracy seen in Figure 3.8a. In an effective training routine, neural losses correlate strongly negative with classification accuracy. The difference in loss between DAB and CSA is within rounding error at 1.50 at a deadline of 150 seconds. DAB is more effective than CSA at meeting deadlines, while maintaining adequate accuracy and loss.

Chapter 4

Multi-Task Data Allocation Under Uncertainty

In this chapter, we present the Minimum Expected Delay (MED) scheme. MED is an uncertainty-aware data allocation scheme in PL that enables multi-task allocation in the presence of uncertainty of learners' characteristics. In MED, the orchestrator allocates resources for multiple learning tasks, each with an associated set of learners that conduct separate global update cycles. Learners are provided with multiple models, one to be trained for each learning task, and the orchestrator determines the amount of data to be allocated to each learner for each learning task. MED allocates data for each learning task among a pool of learners with uncertain computation and communication capabilities. Such uncertainty stems from the fact that the devices used in MEL are user-owned devices and are thus subject to a dynamic user access behavior. Consequently, at any given point in time during the training process of the offloaded task, users can grab their devices and locally run any intensive application/learning task, which causes contention and alters the available learner's computation and communication resources.

4.1 System Model and Overview

Consider a set of n learners $W = \{w_1, w_2, ..., w_n\}$, and a set of m requesters $R = \{r_1, r_2, ..., r_m\}$ that subscribe to the service provided by the orchestrator. The number of learners n is greater than or equal to the number of requesters m. Note that the orchestrator is a centralized entity that is responsible for allocating the computing resources of learners to serve incoming requests. Each request $r_i \in R$ is a parallel learning task that involves training D_i shards of data among the learners such that a certain training deadline T_i is not exceeded. Let Ω_i be the model parameters for the learning task performed for request r_i. Each learner $w_j \in W$ is recruited by the orchestrator in exchange for some incentives, where w_j sets a certain price p_j per data shard, and all requests must be served within a certain budget B. The size (in bytes) of each data shard for each request r_i is denoted α_i. At any given time, each learner w_j can be in one of a set $S = \{s_1, s_2, ..., s_h\}$ of h possible states, each of which corresponds to a potential user activity, such as idle, training, downloading, etc., which triggers different computational and communication capability for the learner. For each learner w_j, each state $s_{j,k} \in S$ triggers different benchmarks $c_{i,j,k}$ and $b_{j,k}$ with a probability $\gamma_{j,k}$, where $\gamma_{j,k}$ indicates the level of certainty that learner w_j is in state $s_{j,k}$. Note that the benchmark $c_{i,j,k}$ represents the compute power of learner w_j for task r_i, which is the rate (in data samples per second) at which w_j can train the model for task r_i when the learner is in state $s_{j,k}$. The benchmark $b_{j,k}$ represents the networking capability (in bytes per second) of learner w_j when the learner is in state $s_{j,k}$. Similar to other works, we assume that the channel is perfectly reciprocal within one global cycle [19].

In MED, it is paramount that each learner uploads the trained parameters of each task r_i to the orchestrator by the deadline T_i. At the deadline, the orchestrator aggregates all parameter sets uploaded to it and restarts the training regime in the next GUC. Learners that do not upload their parameters by the deadline will not be represented in the aggregated parameters that are trained in the next GUC; their work will have been wasted. To improve the performance, we use learner halting. Each learner is aware of the deadline of the task, and periodically checks this deadline throughout their local update routine. If a learner exceeds the deadline, it will halt training and upload its partially trained parameters to the orchestrator, regardless of its progress into the training task. These partial updates are then aggregated as normal, with reduced weighting due to being trained on fewer data samples.

4.2 Problem Formulation

In MED, the main objective is to minimize the sum of the maximum expected delay of all tasks. Note that the delay of a parallel process is the delay of the longest sub-process that needs to be completed. In PL, this means that the time for a GUC is the longest training time rendered by all the learners that are training for the corresponding task [20] [55]. An uncertainty naive approach to minimize the training time for a GUC would be to allocate data to minimize the maximum delay of the learners, as given by Eq. 4.1, where H_i refers to the delay of a GUC for task i, and W_i represents the set of workers associated with task i. Note that the delay for learner j, denoted η_j, can be calculated using the model presented in Chapter 3, as given by Eq. 4.2, where Ω_i refers to the size of the model parameters for task i in bytes, α_i is the size of a data shard for task i, $d_{i,j}$ is the number of data shards allocated to learner w_j, $c_{i,j}$ represents the training rate of learner j in shards/second, and b_j is the network bandwidth of learner j in bytes/second.

$$H_i = \max_{k \in W_i} \eta_k \tag{4.1}$$

$$\eta_j = \frac{2\Omega_i + \alpha_i d_{i,j}}{b_j} + \frac{d_{i,j}}{c_{i,j}} \tag{4.2}$$

The aforementioned model fails to consider uncertainty in learner capabilities. A learner's network bandwidth and training rate depend on the state of the learner. In MED, we account for uncertainty by considering the expected delay given the learners' state probabilities. The expected delay of learner w_j if assigned $d_{i,j}$ data shards from task r_i is given by Eq. 4.3.

$$\delta_{i,j} = \sum_{k \in S} \gamma_{j,k} \left[\frac{2\Omega_i + \alpha_i d_{i,j}}{b_{j,k}} + \frac{d_{i,j}}{c_{i,j,k}} \right] \tag{4.3}$$

As previously mentioned, we aim to minimize the sum of the maximum expected delay for all tasks. Note that the expected delay of each learner is calculated by summing the delay in each state multiplied by the probability of that state. MED strives to achieve this objective while abiding by certain budget and deadline limits. We formulate the data allocation problem as an Integer Linear Program (ILP), where the decision variable $d_{i,j}$ indicates the amount of data shards of request r_i that is allocated to learner w_j. The MED problem formulation is given by Eq. 4.4.

$$\min_{d_{i,j} \in \mathbb{Z}_{\geq 0}} \sum_{i \in R} \max_{j \in W} \delta_{i,j} \tag{4.4a}$$

subject to:

$$\delta_{i,j} \leq T_i, \forall r_i \in R, \forall w_j \in W \tag{4.4b}$$

$$\sum_{w_j \in W} d_{i,j} = D_i, \forall r_i \in R \tag{4.4c}$$

$$\sum_{r_i \in R} \sum_{w_j \in W} d_{ij} * p_j \leq B \tag{4.4d}$$

$$\sum_{r_i \in R} d_{i,j} \geq D_l, \forall w_j \in W \tag{4.4e}$$

$$d_{i,j} > 0 => d_{z,j} = 0, \forall r_z \in R, z \neq i, \forall w_j \in W \tag{4.4f}$$

The objective Eq. 4.4a is subject to the constraints 4.4b-4.4f. Constraint 4.4b ensures that the expected training time of each task r_i does not exceed its deadline T_i. Constraint 4.4c ensures that for each task r_i, the total number of data shards assigned to all learners is equal to the corresponding size of the task's training data set D_i. Constraint 4.4d ensures that the total recruitment cost of all the learners used to train all tasks does not exceed a certain budge limit B. Constraint 4.4e is used to guarantee that each learner is assigned a lower bound of data shards D_l. This is to ensure the utilization of all learners, which can positively affect accuracy. Constraint 4.4f, ensures that each learner is assigned data shards from at most one request.

4.3 Performance Evaluation

In this section, we implement the MED problem formulation in Gurobi and use Axon to facilitate the corresponding multi-task parallel learning algorithm among a set of real devices. We evaluate its performance compared to the MinMax Training Time-Ideal (MMTT-Ideal) scheme and the MinMax Training Time-Uncertain (MMTT-Uncertain) scheme. Note that MMTT-Ideal adopts the ideal case, where the orchestrator has a perfect knowledge of the learners' capabilities at any given state. Though unrealistic, MMTT-Ideal considers the case where there is no uncertainty, and thus acts as an upper bound on the reachable potential of uncertainty-naive schemes. In contrast, MMTT-Uncertain is a representative of uncertainty-unaware schemes, which are typically used in the literature [20] [55]. In MMTT-Uncertain, the orchestrator assumes that the learners' capabilities remain the same during the training process when in fact they do change over time due to the dynamic user access behavior. We use the following performance metrics:

1. **Average Training Time:** The average time it takes to complete the tasks.
2. **Satisfaction Ratio:** The ratio of the number of tasks that finish training before their specified deadline to the total number of tasks.
3. **Average Task Fulfillment:** The number of data samples trained on divided by the total number of data samples in the task's dataset, averaged among all tasks. Learners halting early in their training routine reduces this metric.
4. **Average Data Drop Rate:** The total number of data samples in the task's dataset minus the number of samples trained on, divided by the total number of data samples in the task's dataset, averaged among all tasks.
5. **Average Occupancy Time:** The amount of time the learners spend working on a task divided by the total time to complete the trial, averaged among all tasks. Learners spending more time sitting idle reduces this metric.

Table 4.1: Description of the 3 ML tasks MNIST$_FFN, MNIST_CNN, and Fashion.$

	Architecture	Num Iterations	Deadline	Num Shards
MNIST_FFN	ThreeNN	2	55 (sec)	60
MNIST_CNN	ConvNet	1	40 (sec)	60
Fashion	FashionNet	1	40 (sec)	20

4.3.1 Experimental Setup

We implement MED, MMTT-Ideal, and MMTT-Uncertain on a real testbed of 10 Jetson Nanos, each with 4 GB memory, and a 912 MHz processor. This implementation is done using our custon-built Python framework Axon. Experiments are conducted on three training tasks given in Table 4.1. The architecture of the Neural Network (NN) trained for the three tasks MNIST_FFN, MNIST_CNN, and Fashion is represented by ThreeNN, ConvNet and FasionNet, respectively. Note that ThreeNN consists of three hidden feed-forward layers of size [500, 300, 100]. ConvNet uses two 3×3 convolutional layers with 6 and 16 filters, respectively, then a feed forward layer with 200 units. FashionNet also uses two 3×3 convolutional layers with 16 and 32 filters, respectively, and then two feed-forward layers with 1000 and 50 units each. These neural architectures are described in more detail in Appendix A. All neural networks are trained over 5 GUCs using the Adam optimizer with a learning rate of 0.0001. We use subset benchmarking with a benchmark size of 20 shards to determine learner characteristics. Prior to the experiments, each learner j_w ran network and compute benchmarks for each task r_i and state $s_{j,k}$ to determine $b_{j,k}$ and $c_{i,j,k}$. Unless otherwise specified, the number of learners is set to 10, the number of states is set to 3, the number of tasks is set to 3, the budget is set to 300$, and the minimum number of data shards D_l allocated to each learner is set to 1. We model the variance in the price per data shard specified by learners using a bounded normal distribution. At the beginning of each trial, learners determine their price by sampling a normal distribution of mean 1.0 and variance 0.5. This sample is then bounded to the range [0.1, 1.5].

We simulate variations in learners' capabilities using a set of discrete states, one corresponding to each stressor function. The learner's state distribution is set at the beginning of each trial, and then sampled to determine the state at the beginning of each local training routine. We use three states; idle, training and downloading. The idle state does not involve any stressor function and represents the full capabilities of the learner. The training state represents resource contention with another training task, and is represented via the multiplication of two 900×900 matrices during the training loop. The download state represents contention for network bandwidth and is represented by downloading a 900×900 matrix of random data while the learner is downloading model parameters and data.

We instantiate learner state distributions by feeding a random one-hot vector into a heated softmax function. First, a one-hot vector with elements representing each state is created by sampling a uniform distribution across the states. This vector is then fed into a softmax function, which accepts both a vector and a heat parameter, and outputs a probability distribution across the learner states. The softmax heat parameter controls the spread of the output distribution. High heat values correspond to uniform, high entropy state distributions, whereas low heat values correspond to distributions that are strongly skewed to one state and have low entropy. The resulting state distribution is sampled at the beginning of each training routine on the learner to set the learner's state. In all the experiments describe below, the state heat parameter is set to 0.5.

We model the variance in learner prices using a bounded normal distribution. At the beginning of each trial, learners determine their price by sampling a normal distribution of mean 1.0 and variance 0.5. This sample is then bounded to the range [0.1, 1.5].

4.3.2 Results and Analysis

In our experiments, we evaluate the performance of MED, MMTT-Ideal, and MMTT-Uncertain over varying number of learners, and varying number of tasks. All experiments are repeated 40 times for each instance, and simulation results are presented at a confidence level=95%. Note that the rendered confidence intervals are shown in Figure 4.1. Since the

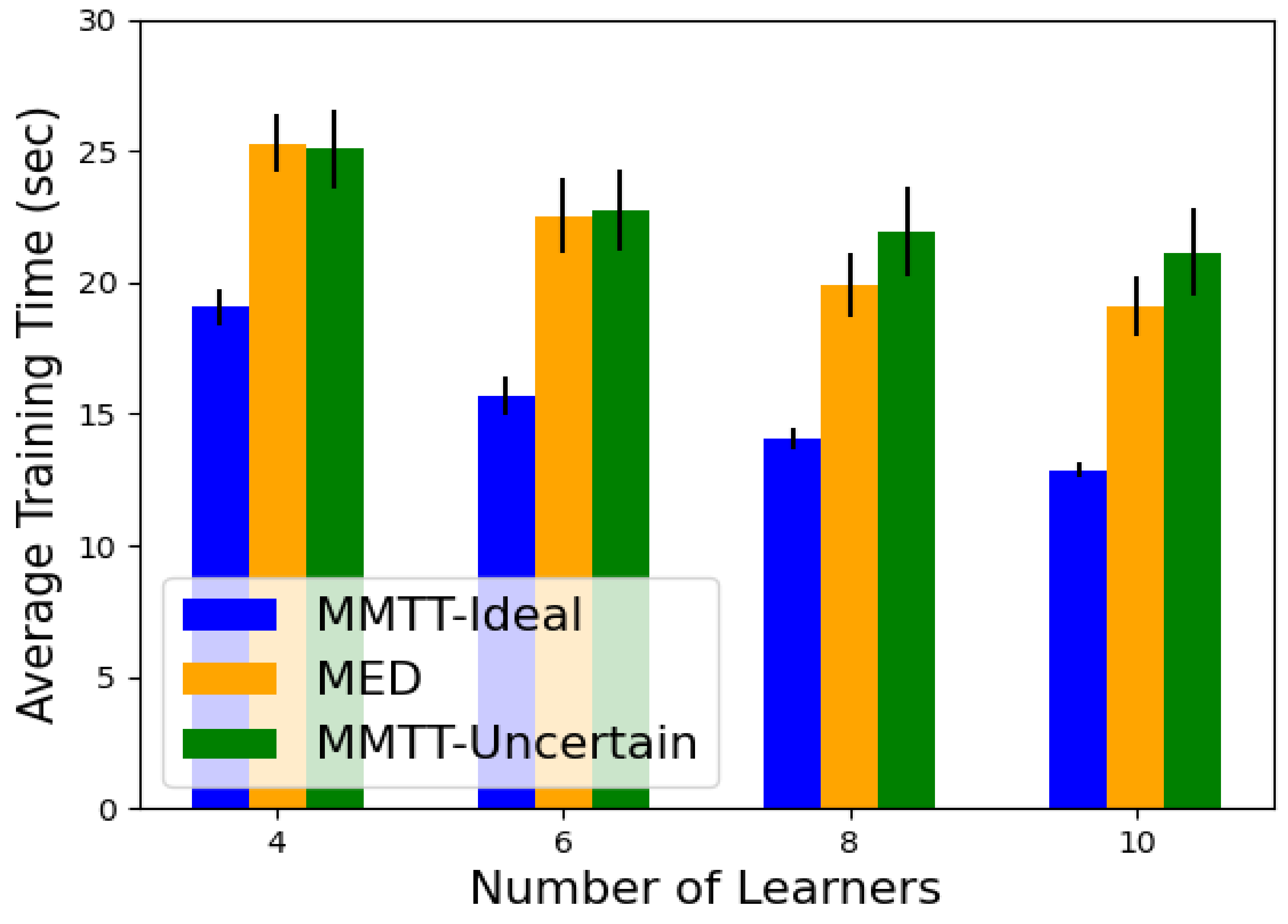


Figure 4.1: The average training time for a global update cycle of MED, MMTT-Ideal, and MMTT-Uncertain over varying numbers of learners.

confidence intervals are negligible, they are not explicitly depicted in the remaining figures for clarity of presentation.

4.3.1.1 The Impact of the Number of Learners

In this experiment, we vary the number of learners from 4 to 10 to assess the performance of MED, MMTT-Ideal, and MMTT-Uncertain under low, medium, and high density of learners.

Figure 4.1 depicts the average training time over varying number of learners. Note that the average training time decreases as the number of learners increases in all schemes. This can be attributed to the fact that as the number of learners increases, the chance of allocating lower amount of data to each learner increases, which increases the learner's speed of training the allocated data, thus reducing the average training time. MMTT-Ideal yields the upper bound on the potential improvement in training time. This is since MMTT-Ideal has perfect knowledge of the learners' states at the beginning of each GUC, and reallocates data to ensure the lowest training time as the learners' states change. MED approaches MMTT-Ideal, with

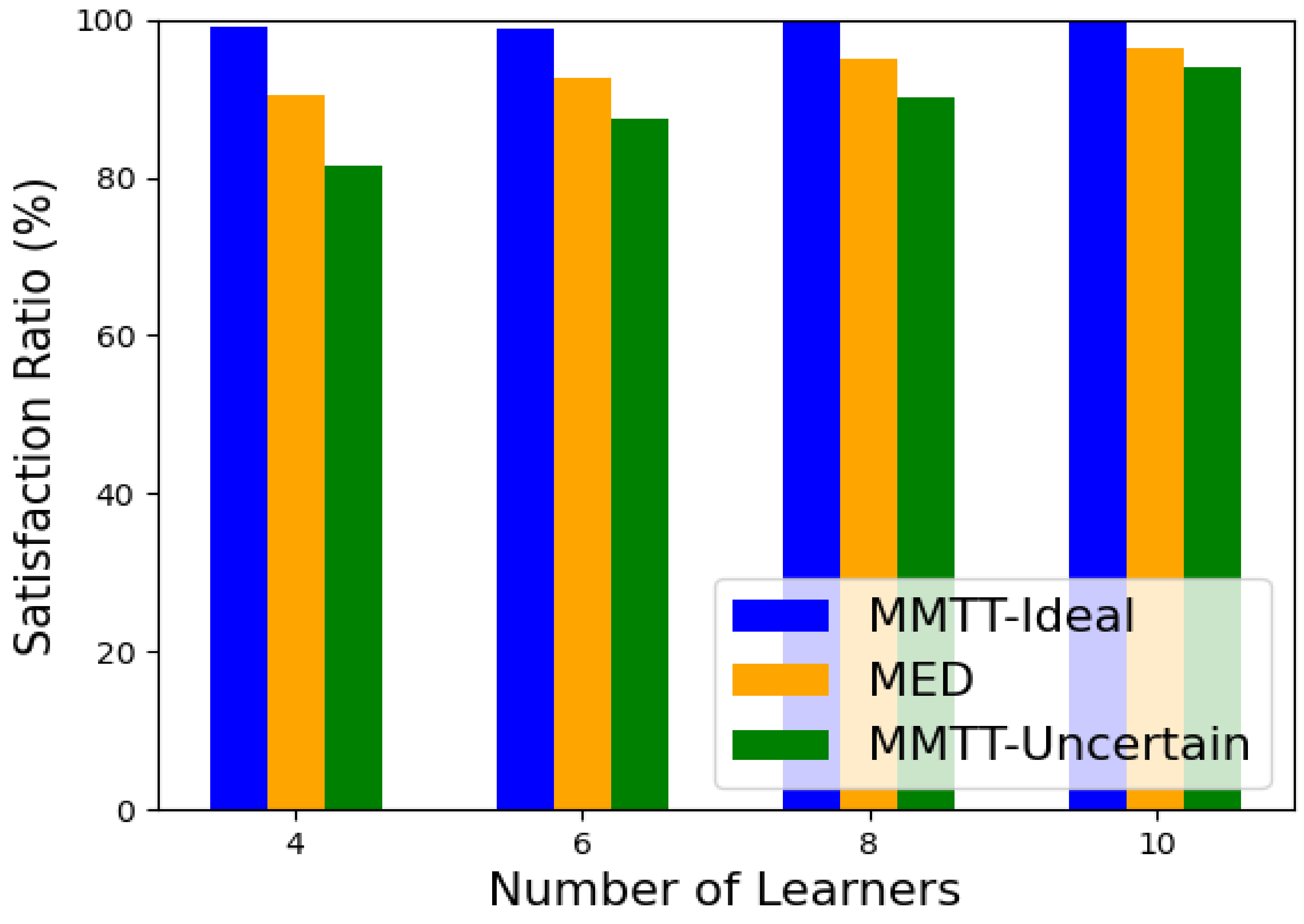


Figure 4.2: The average satisfaction ratio of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of learners.

a gap of up to 32%. In addition, MED significantly outperforms MMTT-Uncertain, with an improvement of up to 11%. This is due to the fact that MMTT-Uncertain does not account for uncertainty and allocates data based on inaccurate benchmark scores that reflect the learners' capabilities when they are in different states than what they are actually in. Thus, there is a high risk of allocating more data to less capable learners, which can prolong the training time in MMTT-Uncertain. In other words, MMTT-Uncertain is much more prone than MED to overestimating the capabilities of resource-poor learners, and underestimating the capabilities of resource-rich learners.

We conduct the same experiment to assess the satisfaction ratio of MED, MMTT-Ideal, MMTT-Uncertain over varying number of learners. As depicted in 4.2, the satisfaction ratio increases as the number of learners increases in all schemes. This is because the higher the number of learners, the fewer the shards of data that are allocated to each learner. Thus, learners tend to complete their training routines faster and are more likely to complete the task before the deadline. MMTT-Ideal has near perfect satisfaction ratio for all cluster sizes.

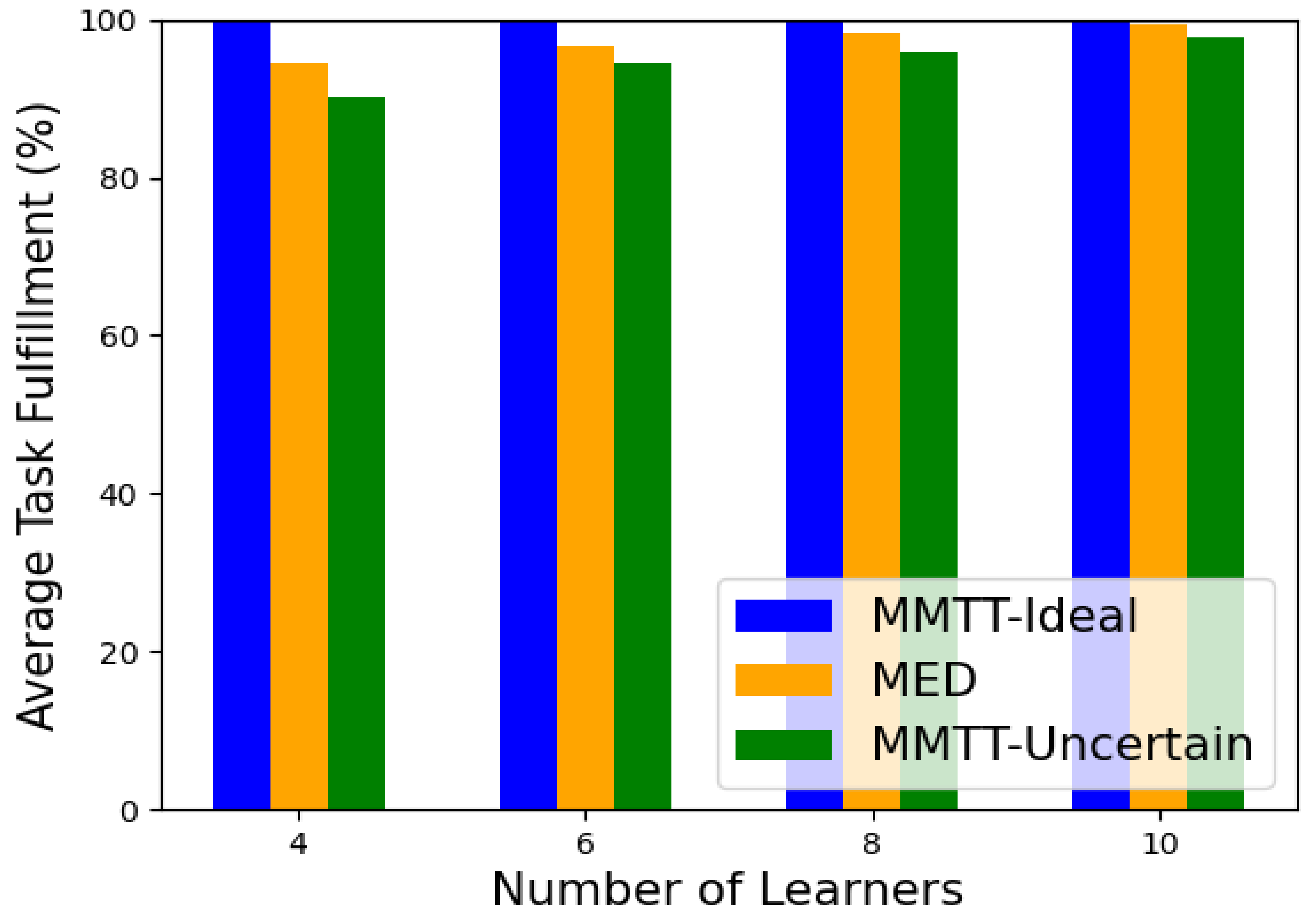


Figure 4.3: The average task fulfillment of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of learners.

This is due to the fact that it has perfect knowledge of the learner states at each GUC and can reallocate to maintain efficiency as the learner states change. MED approaches MMTT-Ideal in satisfaction ratio as the number of learners increases, with a gap of at most 10%. MED also outperforms MMTT-Uncertain with a gap of at most 10%. This can be attributed to the fact that MED has a probabilistic knowledge of the learner states, whereas MMTT-Uncertain assumes that the learner states remain fixed throughout the training process and does not possess any knowledge about their dynamic change. As a result, MMTT-Uncertain tends to over allocate data to weak learners while underutilizing strong learners, leading to a lower likelihood of finishing tasks on time than MED.

Figure 4.3 shows the average task fulfillment of the three schemes over variations in the number of learners. Average task fulfillment is a metric on how far the learners made it through the data that was allocated to them before halting at the deadline. Learners exceeding the deadline and halting will reduce average task fulfillment, and this reduction is weighted by the amount of data that each halting learner was over-allocated. As with

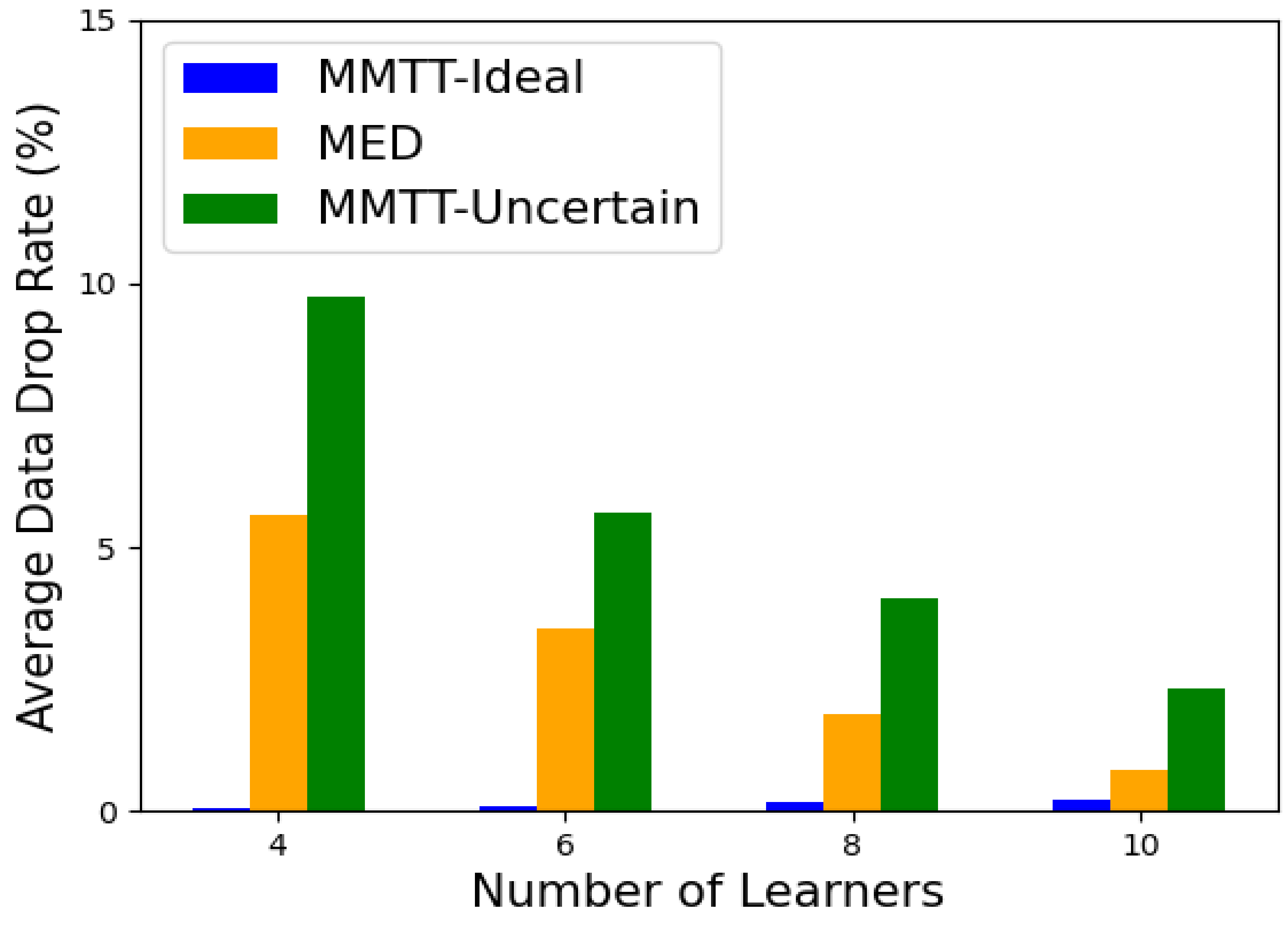


Figure 4.4: The average data drop rate of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of learners.

satisfaction ratio, average task fulfillment improves as the number of learners increases due to there being more resources available for training. MMTT-Ideal consistently has near-perfect average task fulfillment due to its knowledge of learner states. As the number of learners increases, MED approaches MMTT-Ideal in average task fulfillment. The largest gap between MMTT-Ideal and MED occurs at 4 learners, where MED fulfilled on average 4% fewer batches than MMTT-Ideal. MED outperforms MMTT in average task fulfillment, with an improvement of at most 4.3%. This can be attributed to the fact that MED takes the uncertainty in learner capabilities into account during data allocation, and avoids allocating data to learners with a high chance of becoming stragglers. MMTT-Uncertain stands a greater chance of allocating batches to learners that are unable to complete them before the deadline due to being unaware of the uncertainty in learner capabilities.

The impact of varying the number of learners on average data drop rate is shown in Figure 4.4. The average data drop rate is a metric that quantifies the number of data batches that were allocated but not trained on, therefor learners halting early in their training routine

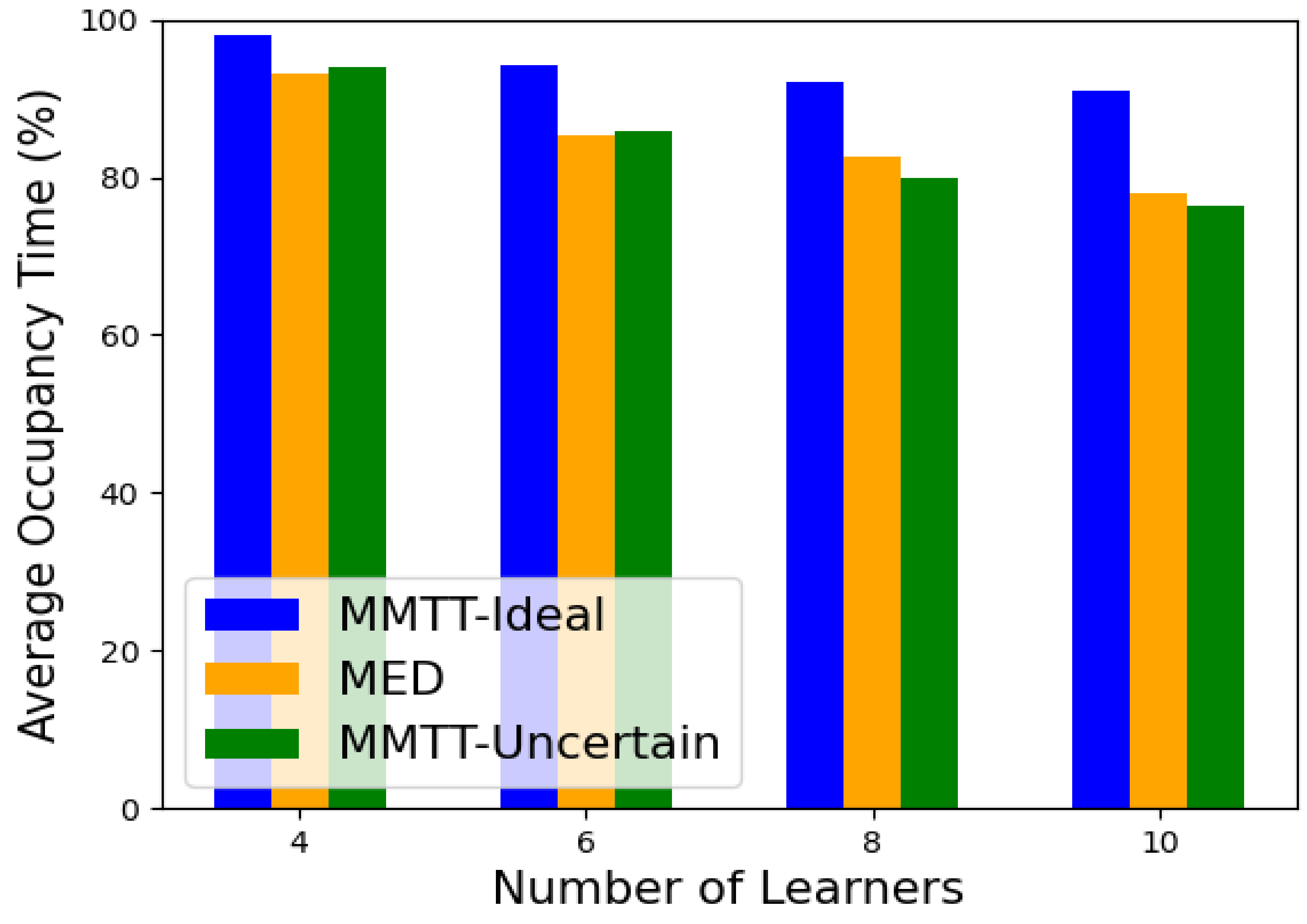


Figure 4.5: The average time occupancy of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of learners.

will increase the average data drop rate. As the number of learners increases, the average data drop rate decreases. This is because with more learners, fewer data shards need to be assigned to each learner, and so the learners are more likely to complete the whole task in the given time. Data drop rate also has an inverse relationship with task fulfillment, and average task fulfillment increases with the number of learners. MMTT-Ideal has a average data drop rate of nearly zero percent for all numbers of learners, due to the fact that it can reallocate data in between GUCs with knowledge of learner states to ensure the whole tasks is completed. MED approximates the performance of MMTT-Ideal as the number of learners increases, with a gap of at most 5%. MED drops fewer data shards that MMTT-uncertain for all cluster sizes, with a maximal performance difference of 41%. This is due to MMTT-Uncertain being unaware of the uncertainties in learner training capabilities, and being more likely than MED to allocate data to learners than have a low probability of completing before the deadline.

Learner average occupancy time is the amount of time learners spend in their local update routine divided by the total amount of time they were associated with the task. Occupancy is a scheduling metric, representing how efficiently each scheme utilized the resources available to them. Figure 4.5 shows that average occupancy time declines for all schemes as the number of learners increases. This is due to the increase in uncertainty as the number of learners increases; with many learners it is more likely that one learner will upload their update late forcing the others to sit idle for some time, wasting resources. As with training time, MMTT-Ideal represents the upper bound, having the best average occupancy time out of the three schemes due to its knowledge of learner states. MED has the second best average occupancy time, with a difference of at most 14% with MMTT-Ideal. MED outperforms MMTT-Uncertain, with the largest improvement being 4.3%. MED outperforms MMTT-Uncertain due to the fact that MED takes into account knowledge of the whole state distribution of each learner, whereas MMTT-Uncertain only accounts for the learner capabilities at the time when it ran benchmarks.

4.3.1.2 The Impact of the Number of Tasks

We now assess the effect of varying the number of tasks while keeping the number of learners fixed at 10. The tasks being trained were the same three tasks as used above, with the exception that each of their deadlines is increased by five seconds, and the budget is set to 200. Tasks are added to the work queue in the order they are listed. Trials that train one task train MNIST_FFN, two tasks MNIST_FFN and MNIST_CNN, and three tasks include Fashion. For four tasks we duplicate the fashion task; the learners train on MNIST_FFN, MNIST_CNN, Fashion, and Fashion_2.

Figure 4.6 shows the average training time as the number of tasks varies. We see that for all schemes, average training time increases as the number of tasks increases, up until around 42 seconds, when the increase in average training time reaches a plateau. Training time increases with the number of tasks due to the increased load on the cluster. Allocating for an increased number of tasks grows the total amount of data that must be allocated to each learner, increasing the amount of time they must spend training. The average training

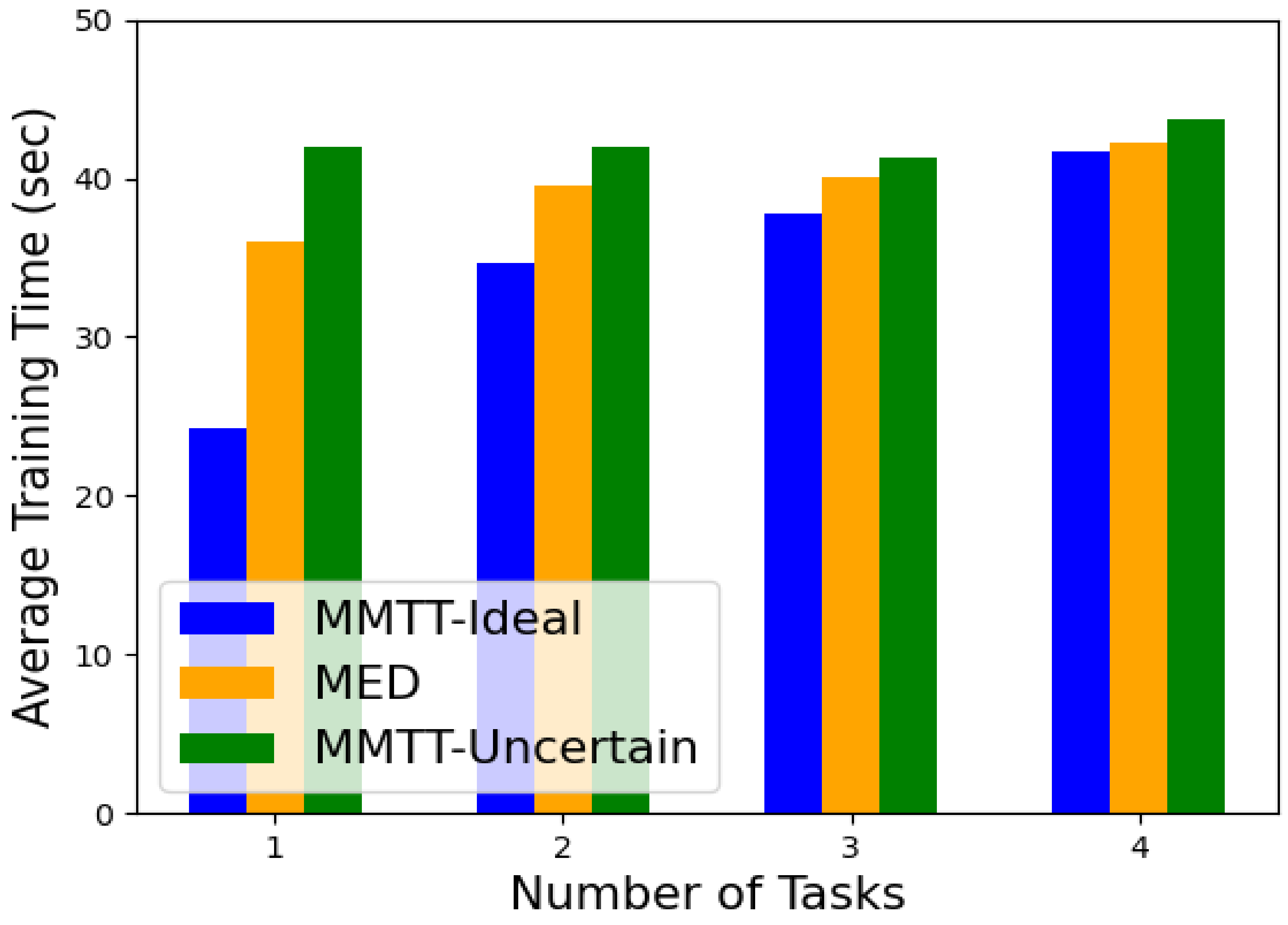


Figure 4.6: The average training time of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of tasks.

time does not increase past around 42 seconds due to learners halting. Learners periodically check the deadline and if they have passed it, will halt training and upload a partial update. This bounds the training time to a short amount of time after the deadline. As with average training time when the number of learners was varied, MMTT-Ideal outperforms both other schemes due to its knowledge of learners states. MED approximates the performance of MMTT-Ideal, being at most 33% slower than MMTT-Ideal. MED is also much faster than MMTT-Uncertain, taking up to 16% less time. This is due to MMTT-Uncertain being inaccurate in its assessment of learner capabilities. Since it does not take uncertainty into account, it does not provision resources as effectively as MED to reduce the training time.

Note that the training time at three tasks is much higher than the training time reported at 10 learners in Figure 4.1. With the same set of tasks, the schemes take around 40 seconds to complete whereas in the last experiment they took around 20 seconds. We attribute this to the higher deadline and lower budget in this experiment.

Figure 4.7 gives the satisfaction ratio as the number of tasks is varied. For all schemes,

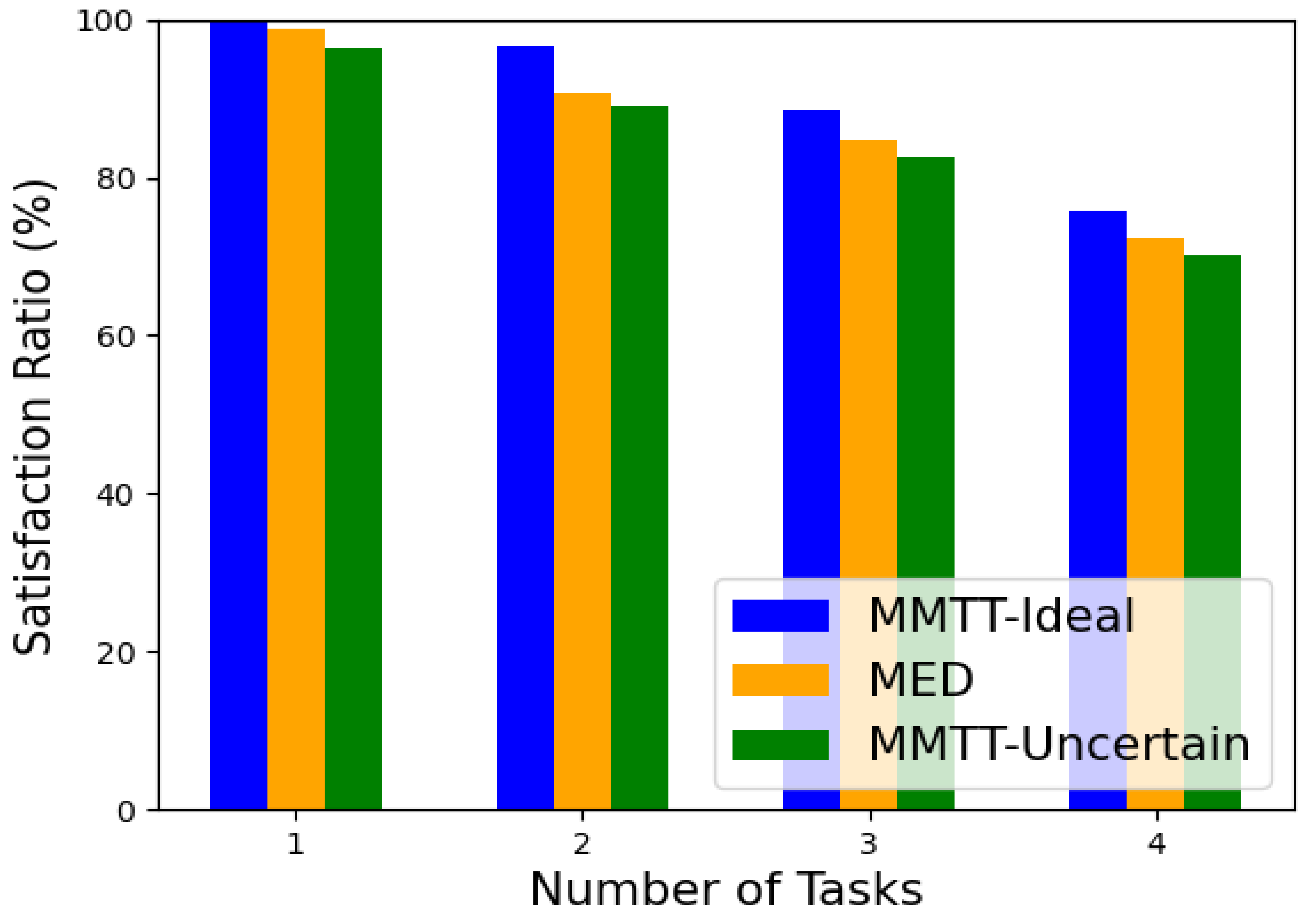


Figure 4.7: The average satisfaction ratios of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of tasks.

satisfaction ratio declines as the number of tasks increases. This is due to the increased load on the system due to there being more tasks and therefor more data shards to allocate. Again, MMTT-Ideal performs the best due to its ability to reallocate data as the states change over time. Even without knowledge of learner states, MED closely approximates the satisfaction ratio of MMTT-Ideal, with a gap of at most 6%. MED also has a consistently higher satisfaction ratio than MMTT-Uncertain, with a difference of at most 4%. This is because MMTT-Uncertain only considers the learner capabilities in a single state, rather than considering the distribution of learner states a MED does. This disadvantage leaves MMTT-Uncertain much more prone than MED to over-allocate data to weak learners, reducing satisfaction ratio.

Figure 4.8 gives the average task fulfillment as the number of tasks are varied. The average task fulfillment drops as the number of tasks increases due to system load. With more tasks each learner must be allocated more data, and so they stand an increased chance of not completely fulfilling their task. MMTT-Ideal performs the best due to its knowledge

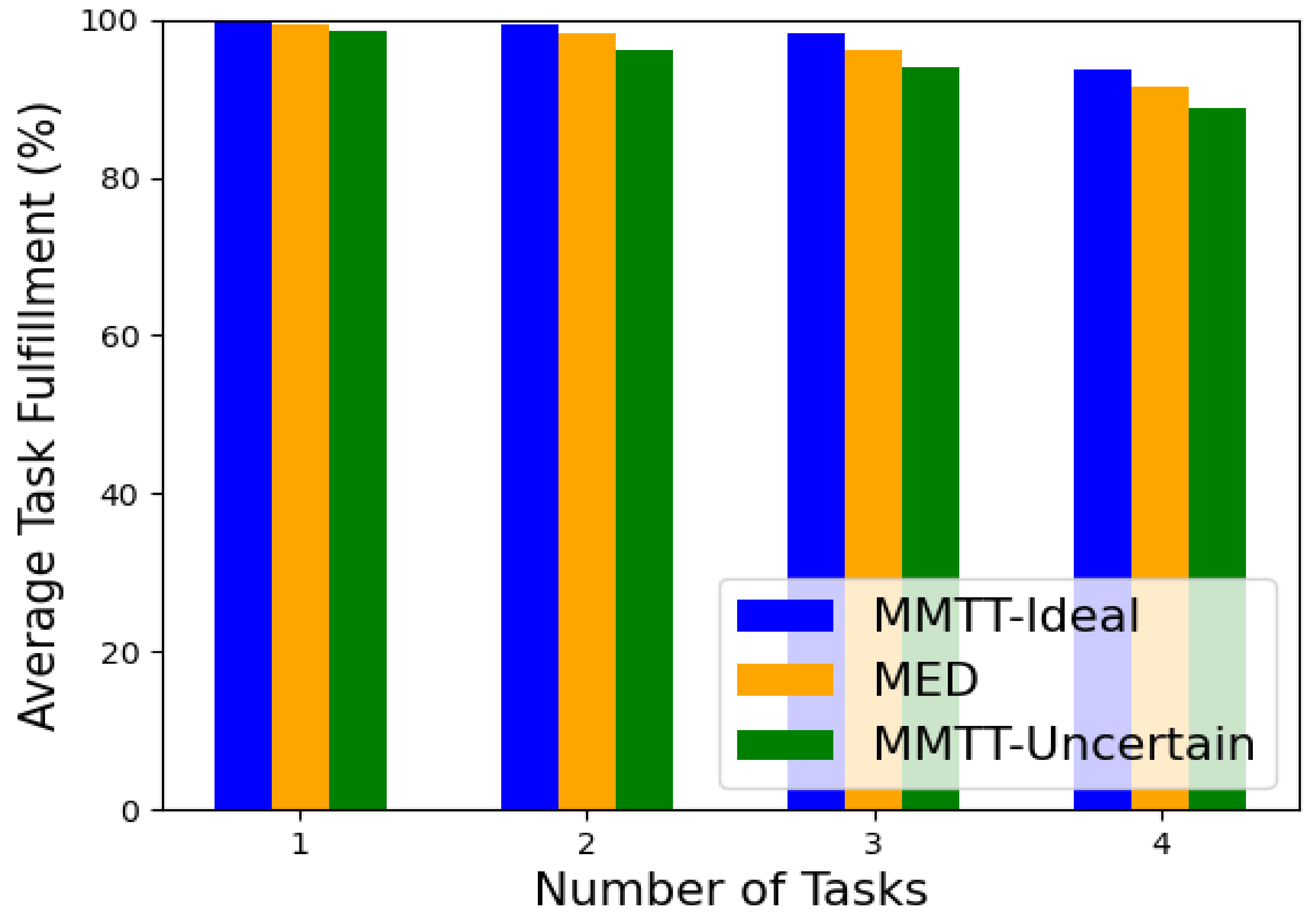


Figure 4.8: The average task fulfillment of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of tasks.

of learner states and ability to re-allocated to ensure fulfillment. Note that at 3 tasks, the average task fulfillment is not perfect for MMTT-Ideal as it is in Figure 4.3. This is due to the lower budget in these experiments. The next best performing method is MED, which approaches the same fulfillment levels as MMTT-Ideal. The largest gap between MED and MMTT-Ideal occurs at 3 tasks, where MED fulfilled 2% fewer data batches than MMTT-Ideal. MED outperforms MMTT-Uncertain in average task fulfillment, fulfilling at most 3% more data batches. This is again due to the fact that MED allocates data by considering the whole state distribution of each learner, while MMTT-Uncertain only considers one state, being much more likely to over-allocate data to learners that have little chance of fulfilling every batch.

The average data drop rate as the number of tasks is varied is Figure 4.9. For all schemes, the average data drop rate increases as the load on the cluster increases with the number of tasks. This is also due to the task fulfillment decreasing. Data drop rate has an inverse relation with task fulfillment, and so lower average task fulfillment at higher numbers of

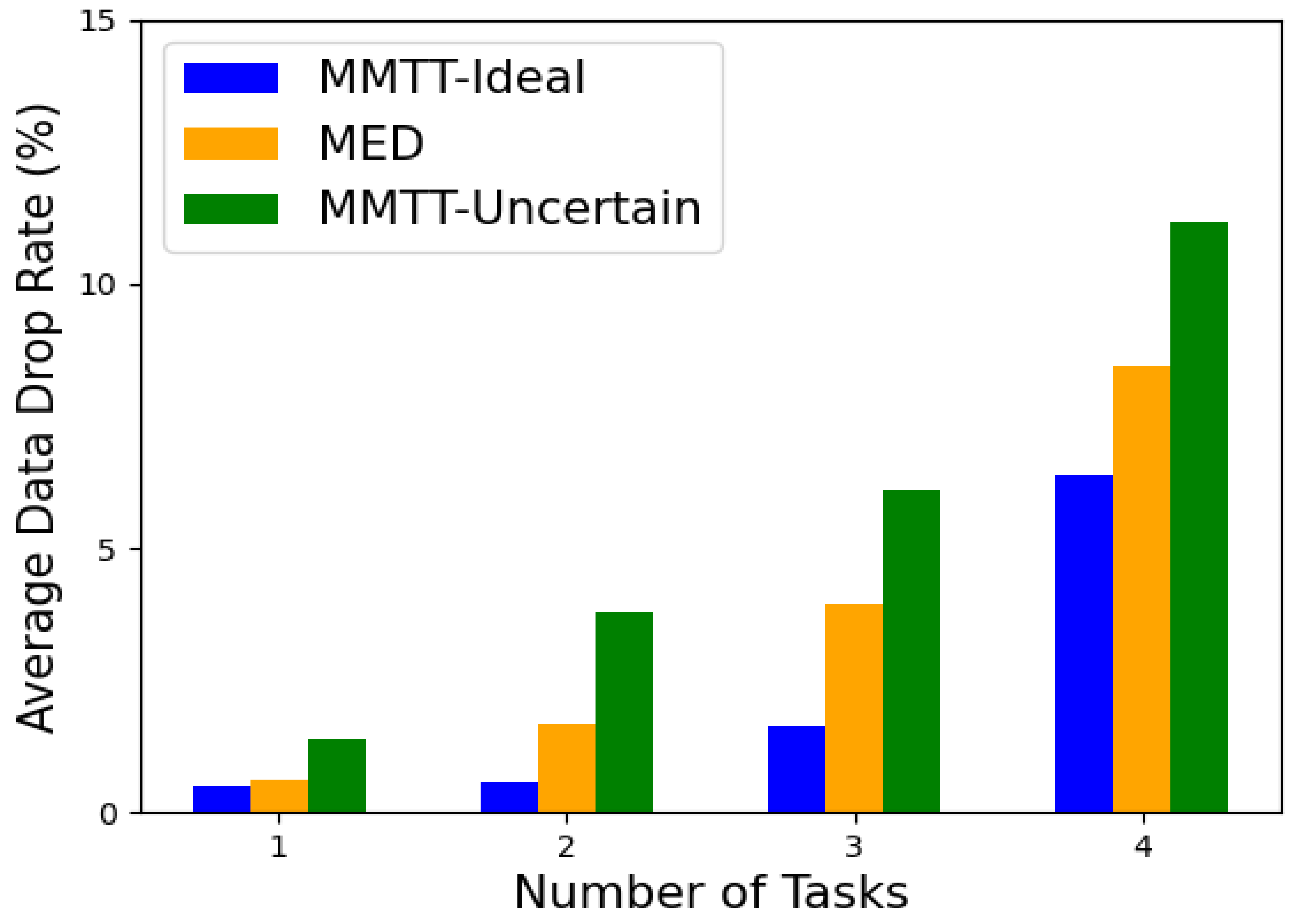


Figure 4.9: The average data drop rate of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of tasks.

task, means average data drop rate will increase with the number of tasks. MMTT-Ideal represents the lower bound on data drop rate, being able to reallocate data in between GUCs using its complete knowledge of learner states to ensure fulfillment. MED approximates the performance of MMTT-Ideal in data drop rate, showing a gap of at most 58% more data batches that are transmitted to learners without being trained on due to learners halting. MED outperforms MMTT-Uncertain in data drop rate, losing 32% more data batches than MMTT-Uncertain. This is again due to MMTT-Uncertain being more likely to over-estimate weak learners, and thus allocate more data to them than they can train on in the given time.

Figure 4.10 gives the average occupancy time as the number of tasks is varied. Occupancy increases with the number of tasks, but like training time it reaches a plateau at around 85%. This is also due to system load causing learners to halt. Occupancy time is calculated from the learners' training times, and so it often follows the same patterns. If a large portion of learners in a GUC are training until the deadline, then the average occupancy time would be high since they all finish at around the same time and can quickly move onto the next cycle

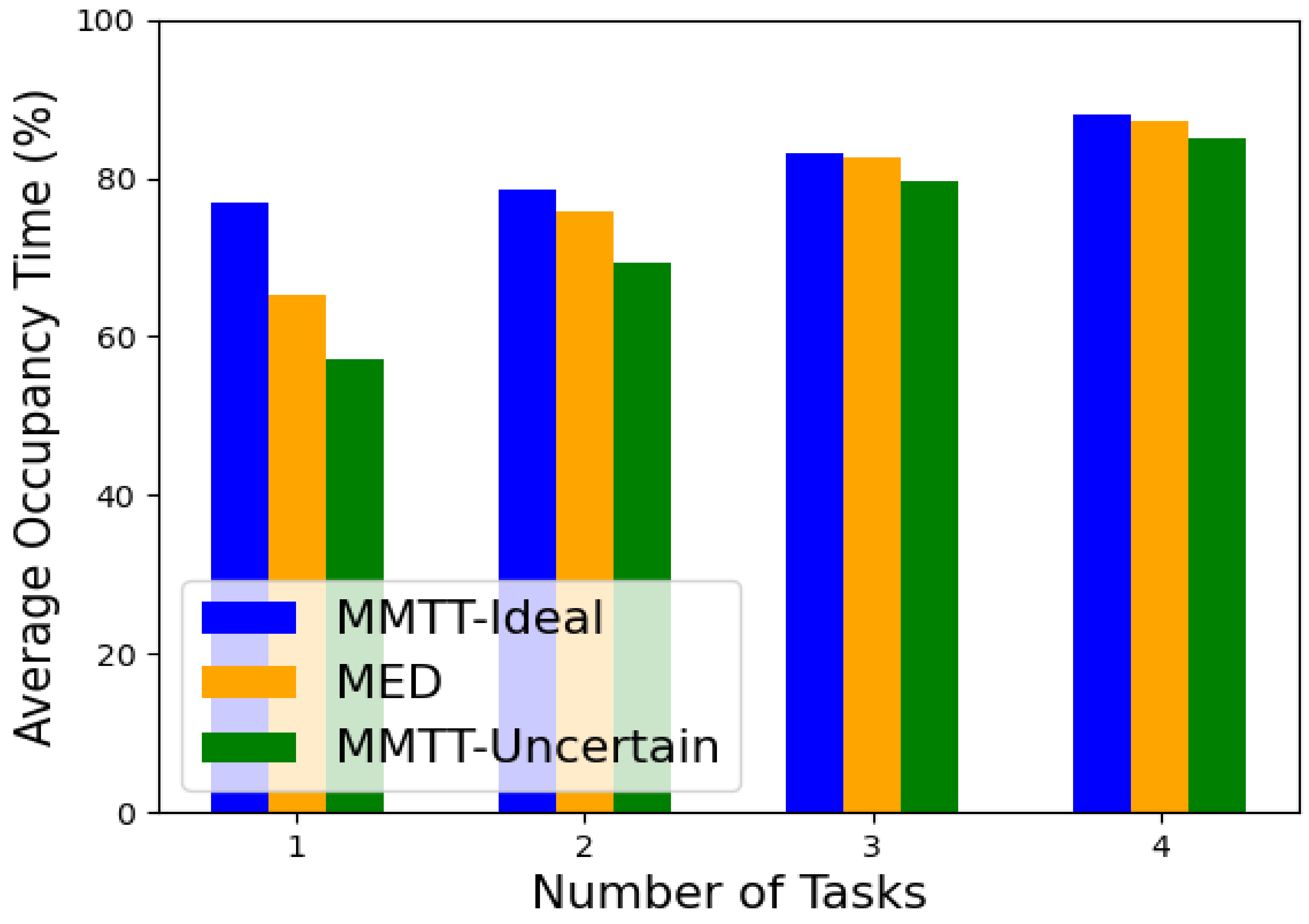


Figure 4.10: The average occupancy time of MMTT-Ideal, MED and MMTT-Uncertain over varying numbers of tasks.

after completing their update. MMTT-Ideal has the highest average occupancy time of the three schemes due to its superior knowledge of learner states. Despite not having knowledge of learner states, MED approaches the performance of MMTT-Ideal in average occupancy time, with a difference of at most 15% at 1 task. MED has better average occupancy time than MMTT-Uncertain at all numbers of tasks, due to it taking uncertainty into account. MED has is more accurate than MMTT-Uncertain at estimating learner capabilities, and so can allocate data with a greater probability that all the learners will finish training at the same time, maximizing occupancy time. MMTT-Uncertain on the other hand, is prone to erroneous estimations of learner capabilities. This means that it is more likely than MED to allocate data such that a small number of learners take a long time to complete while the others finish early and sit idle before the next GUC begins, reducing average occupancy time.

Chapter 5

Conclusion

5.1 Summary and Conclusion

In this work, we have focused on the problem of allocating data in PL. Firstly, we proposed a method to to determine the capabilities of MEL learners with respect to an arbitrary training task. We then developed a decentralized allocation scheme which could ensure synchronicity in heterogeneous environments without centralizing learner characteristics. We then considered the problem of allocating data in the presence of uncertainty in learner characteristics, and found that minimizing expected delay across learner state distributions when allocating data is an effective method to mitigate such uncertainty. All experiments were conducted on real-life systems rather than simulations

We began by proposing SB, which is an extremely simple method to determine the capabilities of a learner for a given task. Our work shows that SB is accurate to within a 1% for predicting the runtime of training tasks and to within 10% for the time to download data and model parameters. SB significantly outperforms the baseline FLOPS by up to 83% in terms of runtime benchmarking error, and its accuracy improves as the benchmark size increases.

We then proposed the decentralized DAB scheme that eliminates staleness in PL by fixing the number of iterations among all learners. DAB does so while preserving the privacy of learners by refraining from sharing any information about their hardware characteristics with the orchestrator. Instead, each learner estimates its own capabilities, and uses these estimations to determine the upper bound on the amount of data that it can train, such that a certain training deadline is not exceeded.

Performance evaluations have been conducted on a real testbed, composed of multiple heterogeneous and distributed devices. It has also been shown that DAB outperforms the baseline CSA in terms of loss and prediction accuracy by up to 12% and 26%, respectively, as the global update index decreases, whereas the latter outperforms DAB by up to 5% and 8%, respectively, as the global update index increases.

We then considered the problem of allocating data for multiple learning tasks in the presence of uncertainty. Our purposed solution, MED, allocates data according to aggregate statistics of learner capabilities. By minimizing the expected delay, MED mitigates the uncertainty in training time, and allocates data more effectively than the uncertainty-unaware baseline.

We extensively assessed the performance MED on a real-life system composed of multiple devices distributed over a LAN. Uncertainty in learner capabilities was modeled using stressor functions, which would place load on system components to contend for resources with the training task. We found that MED reduced training time by 10% and improved data drop rate by 42% as compared to our baseline MMTT.

5.2 Future Research Directions

In this section, we discuss our recommendations for future work on the topic of MEL. We propose the following research directions that would improve the feild of MEL.

- As mentioned previously, most research in MEL focuses on experiments conducted in simulated environments rather than real testbeds. While simulations can accurately model such environmental features as network delay, there will always be unknown issues that could impact the performance of an algorithm on a real system, and thus testing on physical devices is superior. The main obstacle towards doing so for most researchers is not access to edge devices, but the programming difficulty of implementing such algorithms. Regarding this, initiatives like Axon that reduce the programming difficulty of developing MEL systems are important research tools for the future. We propose future research into enabling frameworks such as Axon, and argue that more MEL research should conduct experiments on real-life environments.

- Our work on multi-task parallel learning neglects to consider the case where tasks are submitted and complete at different times. In a more realistic system, tasks would be submitted to the orchestrator as requesters desire to offload them, and the submitted tasks would take a different number of GUCs to complete. These changes would require the orchestrator to dynamically allocate data, rather than setting the associations and allocations once. As tasks complete, learners associated with them would sit idle unless associated with a different task. In order to increase the generality of multi-task PL, future work should consider the case where tasks take different number of GUCs to complete and are submitted to the orchestrator intermittently over some interval of time.

- In PL schemes where learners are paid for the training tasks they perform, greedy learners can charge for more work than they intend to fulfill. Malicious learners could fake high capabilities to be allocated lots of data that they will be paid for, and then train for less than the amount for data they've been allocated while collecting full payment. In this case, the orchestrator would not have any way to detect such an action, there are few ways to tell how much data an set of parameters has been trained on. In order for PL systems such as the ones proposed in this thesis to be implemented in the real world, there needs to be some way to ensure learners are performing the tasks they are allocated.

References

[1] W. Y. B. Lim, C. Nguyen, H. Dinh Thai, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, “Federated learning in mobile edge networks: A comprehensive survey,” *IEEE Communications Surveys Tutorials*, vol. PP, pp. 1–1, 04 2020.

[2] R. Kelly. Internet of things data to top 1.6 zettabytes by 2020. [Online]. Available: https://campustechnology.com/articles/2015/04/15/internet-of-things-data-to-top-1-6-zettabytes-by-2020.aspx

[3] K. Gyarmathy. Iot statistics and trends to know in 2022. [Online]. Available: https://www.vxchnge.com/blog/iot-statistics

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A survey on mobile edge computing: The communication perspective,” *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[5] L. Peterson, T. Anderson, S. Katti, N. McKeown, G. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay, and A. Vahdat, “Democratizing the network edge,” *ACM SIGCOMM Computer Communication Review*, vol. 49, pp. 31–36, 05 2019.

[6] “Homeedge homepage,” 2022. [Online]. Available: ” https://wiki.lfedge.org/display/HOME/Home+Edge+Project, accessed

[7] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, “A survey on task offloading in multi-access edge computing,” *Journal of Systems Architecture*, vol. 118, p. 102225, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762121001570

[8] H. McMahan, E. Moore, D. Ramage, and B. Agüera y Arcas, “Federated learning of deep networks using model averaging,” 02 2016.

[9] J. Konečný, B. McMahan, and D. Ramage, "Federated optimization:distributed optimization beyond the datacenter," 11 2015.

[10] U. Yaqub and S. Sorour, "Adaptive task allocation for mobile edge learning," 04 2019, pp. 1–6.

[11] H. Yu, S. Yang, and S. Zhu, "Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 5693–5700, 07 2019.

[12] K. A. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards federated learning at scale: System design," in *SysML 2019*, 2019, to appear. [Online]. Available: https://arxiv.org/abs/1902.01046

[13] M. Iverson, F. Özgüner, and L. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *Computers, IEEE Transactions on*, vol. 48, pp. 1374–1379, 01 2000.

[14] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based computational performance predictor for deep neural network training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 503–521. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/yu

[15] E. Diao, J. Ding, and V. Tarokh, "Heterofl: Computation and communication efficient federated learning for heterogeneous clients," *CoRR*, vol. abs/2010.01264, 2020. [Online]. Available: https://arxiv.org/abs/2010.01264

[16] R. Tang, A. Adhikari, and J. J. Lin, "Flops as a direct optimization objective for learning sparse neural networks," *ArXiv*, vol. abs/1811.03060, 2018.

[17] M. Ryabinin, E. A. Gorbunov, V. Plokhotnyuk, and G. Pekhimenko, "Moshpit sgd: Communication-efficient decentralized training on heterogeneous unreliable devices," in *Neural Information Processing Systems*, 2021.

[18] T. Li, A. K. Sahu, A. S. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, pp. 50–60, 2019.

[19] U. Mohammad, S. Sorour, and M. Hefeida, "Task allocation for asynchronous mobile edge learning with delay and energy constraints," *CoRR*, vol. abs/2012.00143, 2020. [Online]. Available: https://arxiv.org/abs/2012.00143

[20] B. Yuan, Y. He, J. Q. Davis, T. Zhang, T. Dao, B. Chen, P. Liang, C. Re, and C. Zhang, "Decentralized training of foundation models in heterogeneous environments," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. [Online]. Available: https://openreview.net/forum?id=UHoGOaGjEq

[21] R. Group, "Mobile statistics report 2021–2025,," 2021. [Online]. Available: https://www.radicati.com/wp/wp-content/uploads/2021/Mobile_Statistics_Report_2021-2025_Executive_Summary.pdf

[22] GSMA, "The mobile economy," 2022. [Online]. Available: https://www.gsma.com/mobileeconomy/

[23] M. Gaur and M. Jailia, "Cloud computing data security techniques—a survey," in *Renewable Energy Towards Smart Grid*, A. Kumar, S. C. Srivastava, and S. N. Singh, Eds. Singapore: Springer Nature Singapore, 2022, pp. 55–65.

[24] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," pp. 2131–2165, 2021.

[25] J. Portilla, G. Mujica, J.-S. Lee, and T. Riesgo, "The extreme edge at the bottom of the internet of things: A review," *IEEE Sensors Journal*, vol. 19, no. 9, pp. 3179–3190, 2019.

[26] S. Shaw and A. Singh, "A survey on cloud computing," 03 2014, pp. 1–6.

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

[28] M. S. Allahham, A. Mohamed, A. Erbad, and H. Hassanein, "On the modeling of reliability in extreme edge computing systems," 2022. [Online]. Available: https://arxiv.org/abs/2208.05817

[29] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762118306349

[30] H. Li, Z. Xu, G. Taylor, and T. Goldstein, "Visualizing the loss landscape of neural nets," *CoRR*, vol. abs/1712.09913, 2017. [Online]. Available: http://arxiv.org/abs/1712.09913

[31] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith, "On the convergence of federated optimization in heterogeneous networks," *CoRR*, vol. abs/1812.06127, 2018. [Online]. Available: http://arxiv.org/abs/1812.06127

[32] N. Yoshida, T. Nishio, M. Morikura, K. Yamamoto, and R. Yonetani, "Hybrid-fl for wireless networks: Cooperative learning mechanism using non-iid data," *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pp. 1–7, 2019.

[33] D. J. Mays, S. A. Elsayed, and H. S. Hassanein, “Decentralized data allocation via local benchmarking for parallelized mobile edge learning,” in *2022 International Wireless Communications and Mobile Computing (IWCMC)*, 2022, pp. 500–505.

[34] W. Zhang, S. Gupta, X. Lian, and J. Liu, “Staleness-aware async-sgd for distributed deep learning,” in *International Joint Conference on Artificial Intelligence*, 2015.

[35] T. Nishio and R. Yonetani, “Client selection for federated learning with heterogeneous resources in mobile edge,” *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pp. 1–7, 2018.

[36] C. Xie, S. Koyejo, and I. Gupta, “Asynchronous federated optimization,” *CoRR*, vol. abs/1903.03934, 2019. [Online]. Available: http://arxiv.org/abs/1903.03934

[37] J. Jiang, B. Cui, C. Zhang, and L. Yu, “Heterogeneity-aware distributed parameter servers,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 463–478. [Online]. Available: https://doi.org/10.1145/3035918.3035933

[38] G. Damaskinos, R. Guerraoui, A.-M. Kermarrec, V. Nitu, R. Patra, and F. Taiani, “FLeet,” in *Proceedings of the 21st International Middleware Conference.* ACM, dec 2020. [Online]. Available: https://doi.org/10.11452F3423211.3425685

[39] T. Zhang, L. Gao, C. He, M. Zhang, B. Krishnamachari, and A. S. Avestimehr, “Federated learning for the internet of things: Applications, challenges, and opportunities,” *IEEE Internet of Things Magazine*, vol. 5, no. 1, pp. 24–29, 2022.

[40] U. Mohammad and S. Sorour, “Adaptive task allocation for asynchronous federated mobile edge learning,” *CoRR*, vol. abs/1905.01656, 2019. [Online]. Available: http://arxiv.org/abs/1905.01656

[41] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “Adaptive federated learning in resource constrained edge computing systems,” *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[42] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, “Neural networks with few multiplications,” 10 2015.

[43] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations,” *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[44] Q. Xia, W. Ye, Z. Tao, J. Wu, and Q. Li, “A survey of federated learning for edge computing: Research problems and solutions,” *High-Confidence Computing*, vol. 1, no. 1, p. 100008, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266729522100009X

[45] M. M. Amiri, D. Gündüz, S. R. Kulkarni, and H. V. Poor, “Federated learning with quantized global model updates,” *CoRR*, vol. abs/2006.10672, 2020. [Online]. Available: https://arxiv.org/abs/2006.10672

[46] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Training pruned neural networks,” *CoRR*, vol. abs/1803.03635, 2018. [Online]. Available: http://arxiv.org/abs/1803.03635

[47] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “When edge meets learning: Adaptive control for resource-constrained distributed machine learning,” in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 63–71.

[48] Z. Yang, M. Chen, W. Saad, M. R. Shikh-Bahaei, H. V. Poor, and S. Cui, “Federated learning in 6g mobile wireless networks,” *Computer Communications and Networks*, 2021.

[49] M. S. Allahham, S. Sorour, A. Mohamed, A. Erbad, and M. Guizani, “Motivating learners in multi-orchestrator mobile edge learning: A stackelberg game approach,” *CoRR*, vol. abs/2109.12409, 2021. [Online]. Available: https://arxiv.org/abs/2109.12409

[50] Z. Yang, M. Chen, W. Saad, C. S. Hong, and M. R. Shikh-Bahaei, “Energy efficient federated learning over wireless communication networks,” *IEEE Transactions on Wireless Communications*, vol. 20, pp. 1935–1949, 2019.

[51] U. Mohammad, S. Sorour, and M. Hefeida, “Jointly optimizing dataset size and local updates in heterogeneous mobile edge learning,” 2020. [Online]. Available: https://arxiv.org/abs/2006.07402

[52] C. Witt, M. Bux, W. Gusew, and U. Leser, “Predictive performance modeling for distributed batch processing using black box monitoring and machine learning,” *Information Systems*, vol. 82, pp. 33–52, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437918301765

[53] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, “TBD: benchmarking and analyzing deep neural network training,” *CoRR*, vol. abs/1803.06905, 2018. [Online]. Available: http://arxiv.org/abs/1803.06905

[54] N. Corporation, “Cuda runtime api - event management,” 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html

[55] Qi, E. R. Sparks, and A. S. Talwalkar, “Paleo: A performance model for deep neural networks,” in *ICLR*, 2017.

[56] C. Voskoglou, “What is the best programming language for machine learning?” 2017. [Online]. Available: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7

[57] gRPC contributors, “Supported languages,” 2022. [Online]. Available: https://grpc.io/docs/languages/

[58] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, “A generic framework for privacy preserving deep learning,” 2018. [Online]. Available: https://arxiv.org/abs/1811.04017

[59] Ray Team, “Ray 1.x architecture,” 2020, [Online; accessed 6-January-2023]. [Online]. Available: https://docs.google.com/document/d/1lAy0Owi-vPz2jEqBSaHNQcy2IBSDEHyXNOQZlGuj93c/preview

[60] Deng, Li, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

Appendix A: Neural Architectures

Table 6.1: The ThreeNN neural architecture, by layer

ThreeNN
Linear(784, 500)
ReLU Activation
Linear(500, 300)
ReLU Activation
Linear(300, 100)
ReLU Activation
Linear(100, 10)
Softmax Activation

Table 6.2: The ConvNet neural architecture, by layer

ConvNet
Convolution2D(1, 6, filter_size=(3, 3))
MaxPool(2, 2)
ReLU Activation
Convolution2D(6, 16, filter_size=(3, 3))
MaxPool(2, 2)
ReLU Activation
Flatten
Linear(400, 200)
ReLU Activation
Linear(200, 100)
ReLU Activation
Linear(100, 10)
Softmax Activation

Table 6.3: The FashionNet neural architecture, by layer

FashionNet
Convolution2D(1, 16, filter_size=(3, 3), padding=(1, 1))
MaxPool(2, 2)
ReLU Activation
Convolution2D(16, 32, filter_size=(3, 3), padding=(1, 1))
MaxPool(2, 2)
ReLU Activation
Flatten
Linear(1568, 1000)
ReLU Activation
Linear(1000, 500)
ReLU Activation
Linear(500, 10)
Softmax Activation