AdapTrain: Adaptive Model Partitioning for Efficient Independent Subnet Training on Heterogeneous and Dynamic Cloud Infrastructures

By Mohammadhossein Naderi,

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree of Master of Science

GRADUATE PROGRAM IN COMPUTER SCIENCE

York University
TORONTO, ONTARIO

August, 2025

Abstract

Modern distributed training systems face significant challenges in heterogeneous computing environments, where heterogeneity in computational resources among workers often leads to resource underutilization and extended training durations, particularly in resource-constrained environments. To address these challenges, we propose Adaptive Model Partitioning for Efficient Independent Subnet Training on Heterogeneous and Dynamic Cloud Infrastructures (AdapTrain), a novel framework that dynamically adjusts model partitioning to align with the computational capacities of heterogeneous workers. AdapTrain reduces the overhead of synchronization, thereby minimizing total end-to-end training time by ensuring synchronized completion of training rounds across all workers. Its adaptive design enables robust performance under workload variations, inherent resource heterogeneity, and multi-tenancy effects prevalent in cloud computing environments. An experimental evaluation of production workloads reveals that AdapTrain accelerates model convergence by more than 8x compared to the current training methods. Furthermore, AdapTrain integrates seamlessly into existing systems, introducing negligible system performance overhead while significantly enhancing training efficiency.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Hamzeh Khazaei, for his continuous guidance, support, and encouragement throughout my graduate studies. His invaluable supervision was essential to the development and completion of this thesis.

I also want to thank the members of the Performant and Available Computing Systems (PACS) Lab for the motivating discussions, shared knowledge, and for providing a collaborative research environment during my time in the program.

Finally, I would like to acknowledge my family for their continuous encouragement and support during my graduate studies.

Table of Contents

List of Figures

List of Tables

Declaration of Authorship

I, Mohammadhossein Naderi, hereby declare that this thesis titled, "AdapTrain: Adaptive Model Partitioning for Efficient Independent Subnet Training on Heterogeneous and Dynamic Cloud Infrastructures", and the work presented herein are solely my own efforts.

Chapter 1

Introduction

1.1 Overview and Motivation

Over the past few years, there has been an unprecedented rise in deep learning's popularity, which has become the primary paradigm in vision, natural language processing (NLP), speech recognition, and scientific computing. This is primarily the result of a scaling effect: as both the size of neural networks and the size of the training data grow, the performance of the model grows substantially across a wide set of tasks. This has been investigated in the literature of scaling law, where it is established that performance metrics such as loss and accuracy follow specific trends as the model size and the dataset size increase [1].

An impressive instance of this trend is the GPT-3 language model of OpenAI, a transformer model with 175 billion parameters, trained on more than 300 billion tokens. GPT-3 performed well in few-shot and zero-shot learning tasks, which means that the scale of the model itself can bring a high level of generalization. The transformer architecture became the basis of modern large language models

due to the success of GPT-3 [2]. After that, Meta AI released the LLaMA (Large Language Model Meta AI) family of models. The original LLaMA-1 series with 7B to 65B parameters showed that smaller models could still achieve state-of-the-art performance if trained with high-quality, carefully curated data [3]. This insight was further reinforced in LLaMA-2, which was trained on 2 trillion tokens and showed substantial improvements in both general and instruction-following tasks [4].

The scale effect does not only apply to NLP. Vision Transformers (ViTs) have outperformed convolutional neural networks on standard benchmarks when trained on huge datasets such as ImageNet-21k [5] and JFT-300M [5]. Also, multimodal models such as CLIP and DALL · E have demonstrated that web-scale training on text–image pairs can enhance both vision and language understanding significantly [6, 7]. These developments highlight a shift towards foundation models pre-trained on massive datasets and fine-tuned to downstream tasks with minimal task-specific supervision.

However, this desire to scale is very costly in terms of computation. To give an example, it is estimated that the training of GPT-3 has taken thousands of petaflop-days of compute resources and millions of dollars [2, 8]. Such resource requirements are making participation in state-of-the-art deep learning increasingly exclusive to a handful of tech companies and well-funded labs [9]. Furthermore, the carbon footprint and energy consumption of large-scale training have raised sustainability concerns, encouraging research into more efficient training approaches [10].

On the other hand, the datasets necessary to train such large-scale models have exponentially increased in size. For example, language models are often trained on large-scale corpora such as The Pile [11], a curated dataset containing 825 GB of diverse text, or Common Crawl, which consists of several terabytes of web data. Similarly, massive training sets of images such as LAION-5B [12] and ImageNet-21k are available for training vision models at scale. Processing such huge datasets efficiently requires distributed training, which means that scalability and resource utilization are essential issues in modern deep learning pipelines.

Although distributed deep learning has become a standard approach in high-end datacenters, a new practical challenge has emerged: training in heterogeneous and dynamic settings. In contrast to tightly-controlled and well-managed HPC clusters, in many real-world scenarios (cloud platforms, edge networks, federated learning across organizations, etc.), nodes have different hardware capacities and time-varying loads [13, 14]. As an example, the GPU cluster of a company can include both the older and the newer GPU models with various speeds, or edge devices in a federated setup can range from high-end phones to low-power IoT devices. The links in a network may vary by orders of magnitude in terms of bandwidth (consider inter-region cloud communication, or edge devices over unreliable connections). In this heterogeneous setting, conventional distributed training encounters **straggler** problem: the slowest worker in a synchronous training round dictates the speed of the entire task. The resources are not utilized to the fullest as faster nodes sit idle until slower ones are ready. Such a fundamental bottleneck not only stalls the training progress but also increases the cost of cloud training, as expensive accelerators may spend a significant amount of time

waiting. In addition, static partitioning of data or model between workers can lead to load imbalance. There may be some GPUs that are underutilized while others are overburdened. With model sizes and deployment scenarios increasingly expanding, there is a pressing incentive to develop distributed training methods that adapt to heterogeneous settings and overcome straggler effects. In summary, the emergence of extremely large models has made distributed training a necessity, and the next practical frontier is ensuring that distributed training remains efficient in non-ideal, variable environments where compute and communication resources are neither homogeneous nor reliable.

1.2 Problem Statement

Generally, distributed training of neural networks often relies on synchronous updates used to guarantee correctness. After every mini-batch (or a few mini-batches), workers should synchronize gradients or parameters to ensure that the model remains consistent. The inherent problem with this is that the pace of synchronous training will be as slow as the slowest worker in each round. Even one slow node (due to weaker hardware or transient load) can stall the entire training while faster nodes wait at the barrier. An example of this is traditional data-parallel training that uses All-Reduce. It is based on the assumption that all workers can process their batch in roughly similar time, so that they arrive at the All-Reduce at the same time. Once this assumption no longer holds (as it does in heterogeneous setups), the nominal efficiency of data parallelism fails as well [13].

Common distributed training methods implicitly assume homogeneity. Data

Parallelism typically divides each mini-batch evenly and performs collective communication (All-Reduce or parameter server updates) to aggregate gradients. Frameworks such as Horovod and PyTorch DistributedDataParallel are examples of methods built on this model [15]. They are highly optimized, but they do not take into account differences in worker speed or network bandwidth by default. Each worker gets an equal data shard, and every iteration waits for all shards to finish. Also, typical Model Parallelism schemes presuppose that each partition (layer or a shard of the model) takes a predictable amount of time; a delay in a computation on one partition can stall others. Overall, current common approaches are designed for synchronous, tightly-coupled systems and perform sub-optimally in heterogeneous and non-ideal settings [13].

One recently proposed training paradigm, Independent Subnet Training (IST) [14, 16], offers a degree of decoupling by letting each worker train a subset of the model parameters (i.e., a subnet) independently for a while. After local training, the workers periodically exchange or merge their subnet parameters and form new subnets, and then continue training. IST was introduced to handle cases where distribution is mandatory (e.g., due to memory or privacy constraints). Still, standard data-parallel or model-parallel methods would fail or incur huge communication costs. By training subnets locally and only synchronizing occasionally, IST avoids the per-iteration communication overhead and can tolerate slower networks or strict data privacy (since each worker can keep its data local). However, when applying IST naively to a heterogeneous cluster, one encounters new inefficiencies. Original IST partitions the model evenly into subnets (e.g. each worker gets an equally narrow sub-network) without regard to each worker's capability.

In a heterogeneous cluster, a faster GPU ends up training a subnet of the same size as a slower GPU's subnet, meaning the fast GPU could handle a larger portion of the model or more training steps, but IST doesn't exploit that. Moreover, if all workers must synchronize (exchange subnet parameters) after the same number of local updates, a slow worker could become a straggler at the synchronization point. In other words, IST in its basic form is heterogeneity-unaware. It does not automatically rebalance the work assignment or pace according to each worker's throughput. This can lead to some of the same problems as vanilla data parallelism: faster nodes are underutilized, and the overall training time is dragged down by the slowest node per synchronization round. So, the problem we address is how to retain IST's benefits (less frequent communication, memory efficiency, privacy preservation) while adapting it to heterogeneous environments so that no worker's potential is wasted and the slowest node does not hinder the training process.

1.3 Thesis Contributions

To tackle the above challenges, we propose AdapTrain, a runtime heuristic-based adaptive partitioning method built on IST. At a high level, AdapTrain dynamically adjusts how the global model is partitioned into subnets for each worker, with the goal of balancing the time each worker spends per training round. Unlike prior approaches that might allocate work proportional to hardware specs or static FLOPS estimates, AdapTrain uses online measurements of each worker's actual training speed to continually re-partition the model in response to performance feedback. In essence, faster workers are assigned a larger portion of the model

(more parameters in their subnet). In contrast, slower workers are assigned a smaller portion, such that all workers finish a training round in roughly the same wall-clock time. By equalizing round durations, we eliminate idle waiting time without resorting to asynchronous training. This approach is novel in that it optimizes the balance of work per round. For instance, even if two workers have a 4× speed difference, AdapTrain can partition the model such that the slower one trains, say, a subnet half or a quarter the size of the faster worker's subnet, achieving synchronized completion times, hence minimizing the idle time.

The advantage of AdapTrain's adaptive strategy is a considerable improvement in the efficiency of distributed training on heterogeneous clusters. In our experiments, AdapTrain achieves up to 8× faster training compared to a naive IST deployment that neglects heterogeneity, and similarly outpaces static data-parallel training in heterogeneous environments (details in Chapter 4). These speedups are due to better utilization of all resources: no GPU is left waiting or underutilized. Faster workers simply train larger subnets, contributing more to the overall training without being held back. AdapTrain also maintains high accuracy and convergence stability through careful handling of the hyperparameters, such as the synchronization frequency. In summary, the main contributions of this thesis are:

- We identify a key limitation of IST in heterogeneous environments and demonstrate how this leads to idle time and poor resource utilization.
- We introduce AdapTrain, a simple and robust performance-driven heuristic that adapts the size of subnets allocated to each worker based on the

observed training duration in the previous round.

- We implement AdapTrain as a lightweight, modular framework based on the IST design with minimal coordination overhead.

- We evaluate AdapTrain on two model architectures (FCNN and ResNet-101) deployed on a heterogeneous cluster under synthetic dynamic cloud workloads. Our results show up to 8× faster convergence and significantly improved resource utilization.

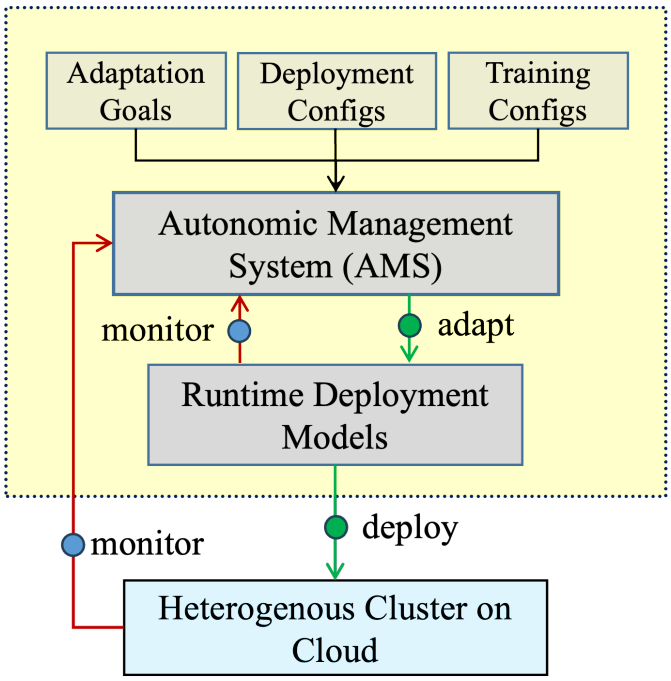


FIGURE 1.1: AdapTrain architecture. A centralized controller collects training round durations, computes updated subnet sizes, and dispatches new partitioning assignments to workers before the next round.

1.4 Thesis Organization

This thesis focuses on improving distributed deep learning in dynamic sub-optimal settings and the presence of system heterogeneity, using the Independent Subnet Training paradigm as a foundation. In Chapter 2, we begin by surveying the landscape of distributed training methods. We present a literature review covering centralized distributed training paradigms (data parallelism, model and pipeline parallelism, hybrid approaches, etc.), and distributed collaborative machine learning methods. We position IST in that taxonomy as a form of partial training with dynamic decomposition. In Chapter 3, we survey the related work that specifically tackles heterogeneous training, and we identify IST's inefficiency in heterogeneous environments. This background justifies the motivation for an approach like AdapTrain. Chapter 4 covers the design of AdapTrain. We outline the adaptive partitioning algorithm, the runtime feedback mechanism, and how we integrate these with the IST workflow. The chapter contains the theoretical arguments and any assumptions behind AdapTrain's design. Chapter 5 provides an extensive experimental evaluation. We implement AdapTrain and compare against baseline methods to evaluate training time, utilization, and convergence behaviour. The results demonstrate where AdapTrain is most beneficial and analyze the trade-offs. Lastly, Chapter 6 concludes the thesis. We discuss the main findings, the implications of our work on training large models in sub-optimal environments, and outline future research directions. Potential extensions (e.g., applying adaptive partitioning to other training paradigms or exploring more complex model architectures) are suggested to inspire subsequent work.

Chapter 2

Preliminaries and Literature Review

Modern machine learning increasingly relies on training data that is distributed across multiple devices. In many real-world scenarios, transferring all raw data to a central server is infeasible or undesirable due to privacy, security, or regulatory constraints. Instead, distributed collaborative machine learning (DCML) techniques allow multiple participants (e.g., user devices, edge servers, or organizations) to jointly train models without exchanging their raw datasets. A prominent example is Federated Learning (FL), wherein a central coordinator (server) orchestrates training across numerous clients that keep their data local. By only sharing model updates (e.g., gradient or weight information) rather than private data, such approaches can protect user privacy while still developing a global model. As a result, FL has been adopted by major technology companies in applications like mobile keyboard prediction, health monitoring, and other privacy-sensitive tasks. This paradigm shift opens new opportunities for learning without sharing raw

data, but it also introduces new challenges in communication efficiency, system heterogeneity, and trust management that do not arise in traditional centralized training.

Overall, distributed training encompasses a spectrum of techniques that differ in how the training job is partitioned across participants. At a high level, two broad scenarios can be distinguished:

- Centralized Distributed Training
- Distributed Collaborative Machine Learning (DCML)

2.1 Centralized Distributed Training

Centralized Distributed Training refers to the classical datacenter or cloud setting where a training job is parallelized across multiple processors or machines that all belong to the same cluster (and trust domain). In this scenario, data can often be freely shared among nodes, and the primary goal is to accelerate training of a large model on a large dataset by splitting the computation. Techniques here include data parallelism, model parallelism, pipeline parallelism, and hybrid strategies. We will review these classical parallelism approaches in this section, as they provide useful context and building blocks.

Training a deep neural network involves finding model parameters (weights) that minimize a loss function on given data. This is typically achieved through iterative gradient-based optimization. During each iteration (or batch), the model performs a forward pass on a set of input samples to calculate predictions and

compare these predictions to ground truth labels using a loss function. Then, the model performs a backward pass (back-propagation) to compute gradients of the loss with respect to all weights. The weights are then updated in the opposite direction of the gradients (scaled by a learning rate). Over many iterations, the model's performance improves as the loss is driven down.

For modern deep networks with millions or even billions of parameters, a single pass through the entire training dataset (i.e., an epoch) may take a very long time on a single machine, so training is often parallelized. Two fundamental forms of parallelism are used:

2.1.1 Data Parallelism

Perhaps the most widely used approach in both cluster and federated settings is data parallelism. In data parallelism, each worker (machine or device) holds a complete copy of the model. The training data is partitioned into shards, and each worker processes a different shard in parallel, computing gradients on its local mini-batches. After each iteration (or after a few iterations), the workers synchronize to aggregate their gradient updates and apply them to the model, ensuring all model replicas stay in sync. Essentially, data parallelism splits the data across workers but replicates the model. This is very effective when the dataset is large and can be easily divided, and when the model size is moderate enough to fit on each worker. The most common approach to perform the synchronization step is a reduce/broadcast (AllReduce) operation to aggregate gradients from all workers and then broadcast the result so that every worker updates to the same new weights. This collective communication can be implemented in a tree or ring

fashion to scale to many workers efficiently.

Data parallel training gains almost linear speedup as the number of workers grows until communication costs or straggler effects become significant. It's also relatively easy to use, as it doesn't require changing the model structure. For example, frameworks such as PyTorch and TensorFlow offer high-level APIs for distributed data-parallel training. Open-source libraries such as Horovod (Uber) encapsulate the AllReduce logic and have demonstrated high scalability (e.g., ImageNet training on 128 GPUs in under an hour). However, a drawback is that every worker must wait at synchronization barriers. If one of the workers is slow (due to weaker hardware or transient load), it slows down the entire training process. Furthermore, each worker must communicate large gradient tensors per round, which can strain network bandwidth for very large models or in network-constrained environments.

The traditional data-parallel approach uses bulk synchronous SGD, wherein all workers synchronize after every iteration. Some systems relax this to mitigate the straggler effect. For example, Hogwild! [17] is an asynchronous training algorithm, in which model updates are performed without locking, so that faster threads can proceed even if others are behind. In distributed settings, the parameter server architecture often supports asynchronous updates. This way, workers can push gradients to a central server and immediately fetch updated weights, without having to wait for other peers. Stale Synchronous Parallel (SSP) [18] is a variant of this approach, which allows a bounded staleness so that a worker can be some number of iterations behind the current global state. Such techniques improve the throughput by not waiting for the slowest worker, at the risk of introducing

gradient staleness (which can affect convergence). To some extent, using these methods can be useful in heterogeneous environments. However, purely asynchronous training can lead to unstable convergence if not carefully tuned, and debugging or reproducing results becomes harder due to nondeterminism.

2.1.2 Model Parallelism

Rather than replicating the entire model on each worker, model parallelism divides the model and partitions model segments across workers. Each worker holds a different portion of the model (e.g., different layers or blocks of a deep network). In a forward pass, data must sequentially pass through portions of the model on different workers. This requires sending intermediate activations between workers along the model's layers. Model parallelism is beneficial when the model is too large to fit in one device's memory (common in very deep networks or large language models). A common example of model parallelism is splitting layers of a neural network among GPUs. For example, GPU_0 processes layers 1-5, then sends the output to GPU_1 for layers 6-10, and so on. One challenge is that if done naively, model parallelism can leave workers idle (e.g., GPU_1 must wait for GPU_0 to finish layers 1-5). To address this, **pipeline parallelism** divides the input mini-batch into smaller micro-batches and pipelines them through the model parts, so all workers are kept busy processing different micro-batches simultaneously. Model parallelism generally involves more complex communication patterns compared to data parallelism, as every forward/backward pass requires inter-worker communication of activations or gradients at partition boundaries. On the other hand, the volume of data exchanged is often smaller than exchanging full gradient vectors for the entire model, and it can be overlapped with computation. Model parallel

strategies can be used together with data parallelism to yield **hybrid parallelism**, which is essential for training massive models (for example, splitting each transformer layer across multiple devices, and also replicating the whole model across node groups for data parallel training).

2.2 Distributed Collaborative Machine Learning

Beyond the aforementioned centralized cluster-oriented parallelism techniques, Distributed Collaborative Machine Learning (DCML) is defined as an umbrella term that refers to all algorithms designed to train models with data distributed across a set of devices without having the training data exposed to any server [19]. In DCML, data remains distributed across parties (often inherently, as each party only has its own local data), and the system is designed such that parties share only minimal information (like model updates or partial computations) to collaboratively train a global model. In this section, we describe several paradigms for collaborative learning in decentralized scenarios with distributed data. These categories differ in what information is shared between participants and how the training process is coordinated. The following subsections provide an outline of key DCML paradigms. The discussion is primarily based on the work by Radovič et al. [19], with focus on aspects that are more aligned with this thesis.

2.2.1 Federated Learning

Federated Learning (FL) enables privacy-preserving model training by offloading the computation of model updates to data-collecting devices, commonly referred to as "clients." In this setup, a centralized server coordinates the training process

by selecting a subset of clients to participate in the current training round, distributing the current global model to these clients, and aggregating the updated models received from them after local training. Through this iterative process, the server improves the global model, which is then used in the following rounds of training [20, 21, 22]. Formally, the FL objective is to minimize the function in Eq. 2.1, where $|\mathcal{K}|$ is the set of devices, each device $k \in \mathcal{K}$ has a training set $\mathcal{D}_k$, and w is a model that fits data in a distributed setup by minimizing some loss function $F(\mathrm{w})$.

$$\min_{\mathrm{w}} \; F(\mathrm{w}) := \sum_{k=1}^{\mathcal{K}} \frac{|\mathcal{D}_k|}{\sum_{j=1}^{\mathcal{K}} |\mathcal{D}_j|} F_k(\mathrm{w}) \tag{2.1}$$

Federated Averaging (FedAvg) [20] is the most prominent algorithm used for solving Eq. 2.1 by applying an SGD optimizer on local functions F_k using the same learning rate for all devices. FedAvg is regarded as a variant of SGD with several global rounds, each of which has a few steps of local update followed by model synchronization. The FedAvg algorithm and its variants [21, 23] require clients to upload and download the entire model during each server round. Although these methods remain popular as baselines, they present two major challenges. First, they cause significant communication overhead by sending large volumes of data over the network. Second, they impose a fixed model architecture on all clients, regardless of their available resources. This can lead to increased carbon emissions [24], fairness issues, and reduced model accuracy [25, 26, 27]. These problems are even more pronounced in real-world DCML environments, where

system heterogeneity raises serious concerns about the effectiveness of parameter-sharing approaches [24, 28, 29]. As a result, some researchers have argued that sharing parameters for knowledge transfer is a flawed design choice [30].

2.2.2 Federated Knowledge Distillation

Knowledge Distillation (KD) was originally proposed as a technique for transferring knowledge from a large "teacher" model to a smaller "student" model [31, 32], which enables deployment on devices with limited computational and memory resources. KD has recently found new applications in DCML. In particular, it has led to the development of Federated Knowledge Distillation (FKD) algorithms, which replace the exchange of model parameters with the exchange of logits between clients and the server. This shift offers several advantages: it significantly reduces communication overhead, supports flexible and heterogeneous model architectures tailored to each client's capabilities [33], and enhances privacy and robustness by avoiding direct parameter sharing [30].

The primary challenge in applying KD to DCML is that it relies on shared data. For effective knowledge transfer, both teacher and student models must be evaluated on the same data samples. However, this violates the core principle of DCML: clients are not allowed to share raw data with one another. FedKD [34] and FML [35] offer a straightforward approach for incorporating KD into DCML. In these methods, each client simultaneously trains its own private model alongside a small, globally shared model using codistillation [36]. The shared model, trained collaboratively across clients, serves as a source of knowledge to guide the

training of the private models. Although this strategy enables architectural flexibility, it also introduces considerable computational overhead, since clients are required to train two models in parallel rather than just one. An alternative approach for applying KD in DCML involves representing teacher knowledge with a fixed set of class-specific vectors, often referred to as "prototypes." As an example, in the Federated Distillation (FD) algorithm [37], the server computes the average logits for each class based on the outputs from all clients. During local training, each client then aligns its predictions with these global class logits by penalizing deviations from them. This method significantly reduces communication costs, often by several orders of magnitude. However, it comes at the cost of accuracy, with performance drops reaching up to 25% compared to the standard FedAvg algorithm [37]. The FedGKT algorithm [38] introduces a novel approach to server-side model training without requiring parameter exchange, wherein clients begin by training a lightweight model split into two parts: an encoder that transforms raw data into intermediate embeddings, and a classification head that produces final predictions from these embeddings. Once local training is complete, clients send their intermediate embeddings, final logits, and ground-truth labels to the server. Using this data, the server trains a larger model that focuses solely on the classification head, learning to map embeddings to target labels. In essence, the server builds a more complex prediction layer than the one used on the clients. To enhance convergence, both the server and the clients apply KD during training. Despite its novelty, FedGKT has limitations: clients must expose their label distributions, and the server-side model remains incomplete, as it lacks the encoder component, which only exists on the clients.

FKD methods can benefit from using an external dataset that reflects the characteristics of the clients' private data. Cronus [30] uses this idea by distributing a public dataset to all clients. At the end of every training epoch, clients share the logits corresponding to a subset of public data samples, which the server then aggregates. During local training, each client uses its own labeled data alongside the public dataset, now labeled with the aggregated global logits. This dual training strategy helps improve model performance while maintaining data privacy. In this algorithm, the server is mainly responsible for lightweight coordination tasks such as aggregating logits or synchronizing training rounds. In contrast, some model-sharing approaches take advantage of the server's computational power to play a more active role in improving the training process. For example, FedDF [39] supports training multiple architectures simultaneously, where fine-tuned client models serve as teachers and aggregated global models act as students during a KD phase on the server. Fed-ET [40] takes a similar approach but allows clients to select from a predefined set of architectures that share a common classification head. After each round, the server generates pseudo-labels for a public dataset using a consensus of client predictions. These labels are then used to train a larger server model. To prepare new models for the next round, Fed-ET combines the averaged lower layers of the client models with the server's classification head. Building on this idea, FedAUX [41] extends FedDF by introducing certainty scores that reflect how closely a client's local data resembles the global dataset. These scores are used to weight client logits during server-side KD and are derived by training a logistic regression model to distinguish between local and global data samples. Despite their innovations, FedDF, Fed-ET, and FedAUX share common drawbacks: they require clients to choose from a fixed set of model architectures

and involve transmitting full model updates. The FedGEMS algorithm [42] introduces a two-way knowledge distillation process involving both the server and the clients. Clients begin by generating logits for each example in a shared public dataset and sending them to the server. The server aggregates these predictions to create pseudo-labels, which are then used to train a global model. In parallel, selected clients also perform KD locally by receiving the server-generated pseudo-labels and aligning their predictions with them, following a process similar to FedMD. Building on this idea, the MHAT algorithm [43] enhances server-side training by combining ground-truth labels with client-generated logits to improve the quality of the server model.

The approaches discussed so far assume that all clients receive the same information from the server, such as uniform global logits. However, this assumption can cause problems when client data distributions vary significantly, as it could lead to degraded model performance. To mitigate this, the KD-pFL algorithm [44] introduces a data similarity matrix that captures how closely related each pair of clients is. This matrix is used to compute personalized logits for each client by calculating a weighted average of other clients' logits, where the weights are determined by the similarity values, while the similarity matrix is trained on the server using a KD-based procedure. The COMET algorithm [45] uses a different approach by combining FKD with client clustering methods [46, 47, 48] to personalize knowledge transfer. In this approach, the server groups client-generated logits using the K-Means clustering algorithm. The resulting centroids are sent back to the clients, who select the centroid that best matches their own logits and use it for local KD during training.

In summary, despite their potential, FKD algorithms still face several unresolved challenges:

- **Model Heterogeneity:** Although many FKD methods are designed to support a wide range of model architectures, empirical evidence shows that allowing some clients to use overly simplistic models can negatively affect convergence across all models [33].

- **Architecture Selection:** There is currently little guidance on how clients should choose their model architectures independently. The implications of these choices in real-world deployments are still largely unexplored.

- **Stateful Training:** Algorithms that do not build a fully functional server-side model require clients to retain state across multiple rounds. In other words, clients must resume training from where they left off rather than starting from scratch. While this is possible in cross-silo FL with a small, stable client population, it becomes infeasible in cross-device FL where participation rates are extremely low and inconsistent [49].

- **Dependence on External Datasets:** Methods that aim to build a usable server-side model typically rely on access to a public or shared dataset. However, finding a representative and centralized dataset may not always be possible. Worse, if the external dataset differs significantly from the clients' data distributions, model performance may suffer [39].

2.2.3 Split Learning

Split Learning (SL) [50, 51, 52] is a distributed training approach where a neural network is divided horizontally across two or more participants. In its basic form, SL involves just two nodes: a client that holds the training data along with the first k layers of the model, and a server that maintains the remaining $N-k$ layers, where N is the total number of layers in the network. This form of SL with only two nodes is technically equivalent to pipeline-parallel training.

Once both sides initialize their respective model segments, the training process begins with the client performing a forward pass on its local data and sending the resulting intermediate representations, known as "smashed" embeddings, along with the corresponding labels to the server. Then, the server completes the forward pass, calculates the loss, and updates its part of the model by backpropagating the error to the first layer. The server then sends the gradient information back to the client, which will be used to perform backpropagation on its own segment. By splitting the model this way, SL allows clients to participate in training larger models than they could handle on their own, all while preserving data privacy. The basic two-node Split Learning setup can be extended into more complex architectures. One notable variation is the "U-shaped" architecture [52, 53], where the client is responsible for both the initial and final layers of the model. This design has the advantage of preserving label privacy, as the client can compute the loss locally without sharing target values with the server. However, it comes with increased communication overhead, requiring four passes per training step: two during the forward pass and two during the backward pass.

When multiple clients participate in collaborative model training, there are

several possible coordination strategies. One approach, proposed by Gupta et al. [50], involves clients training in a round-robin fashion. Each client trains its portion of the model and then passes the updated parameters to the next client in line. While simple, this method has notable disadvantages. Since only one client is active at a time, it results in inefficient resource utilization and slower convergence. In addition, in the presence of statistical heterogeneity across clients, this sequential training can lead to "catastrophic forgetting", where the model loses previously learned knowledge as it adapts to new client data [54, 55].

To address the issue of low resource utilization in SL, the SplitFed algorithm [56] integrates SL with model-sharing techniques by allowing clients to train their local model segments in parallel and then periodically averaging them, following the FedAvg strategy. SplitFed is introduced in two variants. In *splitfedv1 (SFLV1)*, each client maintains its own server-side model, which is averaged with others at the end of each round—effectively making this version functionally similar to FedAvg. In contrast, *splitfedv2 (SFLV2)* assumes a shared server-side model across all clients, which is updated collaboratively. Building on this line of work, the SFLG algorithm [57] generalizes the design by allowing the number of server-side models to vary between one and the total number of clients. Another extension, *splitfedv3* [58], keeps the client-side model private and only aggregates server-side updates. This design helps mitigate catastrophic forgetting [59], but it also comes with a trade-off: it prevents the development of a public client-side model that could be served to new clients.

SL can significantly reduce communication overhead compared to model-sharing approaches in Federated Learning, especially when training large models or when

there are many clients involved [60]. However, SL presents a different challenge: it requires network communication for every batch of data, which can become a bottleneck. The FSL algorithm [61] was proposed to address this challenge. In the FSL algorithm, clients attach a shallow auxiliary head to compute a local loss that will be used to update the client-side model. In this approach, the server calculates its own loss to update only its part of the model, without having to send gradients back to the clients. A similar idea is used in the AdaSplit algorithm [62], where client models are also trained using locally computed losses. However, unlike FSL, in AdaSplit, clients train locally for a few rounds before sending sparse updates to the server. In both approaches, using local loss functions allows client-side training to proceed without relying on gradients from the server.

To summarize, Split Learning (SL) is a class of DCML algorithms, different from the Federated Learning paradigm. SL enables collaborative training across clients with limited memory or computational power by offloading a part of the model computation to a server. While promising, SL still raises several important research questions:

- **Privacy:** SL exposes certain privacy risks, particularly through the transmission of target labels and the potential for the server to reconstruct raw data from intermediate embeddings. The U-shaped architecture can mitigate the label-sharing issue, but protecting against data reconstruction remains an open challenge [63].

- **Scalability:** Because each client consumes server resources such as memory and compute time, SL is currently more practical in small-scale settings.

There is a need for further research to assess its scalability and develop mechanisms to reduce server load as the number of clients grows.

2.2.4 Partial Model Training

Partial Training (PT) is a family of model-sharing Federated Learning methods that relax the assumption of model homogeneity across clients. Rather than requiring each client to train the entire global model, PT allows clients to train smaller sub-models tailored to their capacities. These sub-models are constructed by dropping neurons from fully connected layers or excluding filters in convolutional layers [19]. Fig. 2.1 demonstrates the difference between model-sharing algorithms, Split Learning, and Partial Training in terms of the model partitioning strategy across clients.

In Partial Training, clients are unaware that the model they receive is only a subset of a larger global model maintained by the server. The server is only responsible for coordinating the training process across all clients. During each epoch, the server assigns a specific portion of the global model to each participating client and provides the corresponding weights needed for training that segment. Once the sub-models are extracted, the server sends each one to the corresponding client. Clients then train the received sub-model locally and return the updated parameters to the server after completing their training. The server then performs aggregation using its custom aggregation logic based on how the sub-models were constructed in the first place.

Together, the decomposition and the aggregation functions define the core behavior of any Partial Training (PT) algorithm. Based on how these two components are defined, PT methods can be classified into three main categories: static, dynamic, and independent subnet training (IST).

Partial Training with Static Decomposition

In static decomposition, the server generates sub-models in a deterministic and consistent manner across all training rounds and clients. The structure of each sub-model is completely determined by the current metadata (e.g., available memory, computational resources, local data characteristics, etc.) associated with the client, independent of any additional algorithm-specific state (e.g., sub-model construction strategy, model parameters, etc.) from previous rounds. This approach is deterministic in how sub-models are constructed. However, it does not mean that a given client will receive the same sub-model in every round, as the client's metadata may change over time.

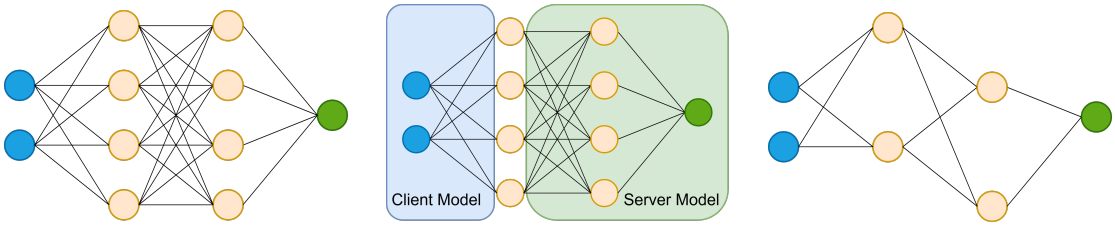
Two notable examples of PT with static decomposition are the HeteroFL [64] and FjORD [65] algorithms. In HeteroFL, the server defines a set of model capacity levels and constructs corresponding sub-models by adjusting the network width, for example, by modifying the number of channels in convolutional layers or the number of neurons in fully connected layers. These sub-models are organized hierarchically, such that a smaller sub-model is always a subset of a larger one. This creates a nested structure where smaller sub-models are fully embedded within larger ones. FjORD follows a similar approach by introducing a hierarchy of nested sub-models using a method called Ordered Dropout, which ensures that

smaller models are consistently contained within larger ones.

Partial Training with Dynamic Decomposition

A key limitation of Partial Training algorithms with static decomposition is that they can only support the training of models of up to the size of the largest model that could fit on clients. In contrast, PT algorithms with dynamic decomposition overcome this limitation by progressively training different parts of the global model over time. This can be accomplished either through pseudo-random selection of sub-models or by applying heuristics that guide which components of the model should be trained in each round.

One example of pseudo-random sub-model construction is the Federated Dropout algorithm [66, 67]. In this method, the server maps each client's metadata to a single scalar value between zero and one, which determines the fraction of the model to be used. It then randomly selects that fraction of neurons from fully connected layers and channels from convolutional layers to form the sub-model. However, recent studies have highlighted the limitations of this approach and questioned its effectiveness [68]. FedRolex [69] and PriSM [70] are two examples of Partial Training with heuristic-based sub-model construction. In FedRolex, the server uses a rolling strategy in which a sliding window moves across the global model to determine which parts to include in each sub-model. This ensures that different sections of the model are trained over time. In contrast, PriSM constructs low-rank sub-models by applying singular value decomposition, allowing the server to extract compact yet informative representations of the global model for training.



(A) (B) (C)

FIGURE 2.1: (A) In model-sharing algorithms, the entire model is sent to every participating client for local training. (B) In Split Learning, the model is divided horizontally across layers, with different segments handled by the client and the server. (C) In Partial Training, the model is partitioned vertically, meaning each client trains a smaller but complete version of the model that can be trained end-to-end. Adopted from [19].

2.3 Independent Subnet Training

Independent Subnet Training (IST) [14] is a special case of Partial Training with dynamic decomposition, where the global model is randomly decomposed into disjoint non-overlapping subnets. This approach significantly simplifies the aggregation process: for any parameter that was updated during the current training round, the server simply copies the updated value into the global model, and parameters that were not trained simply keep their previous values.

IST was originally introduced for fully connected networks [14]. In such cases, each hidden layer is divided into equally sized neuron partitions, and subnets are constructed by selecting only the weights that connect neurons within the same partition, as depicted in Fig. 2.2. This creates disjoint sub-networks that can be trained independently without requiring cross-subnet communication. Workers train their assigned subnets for a few iterations (i.e., a training round) before synchronizing their updates and aggregating them into the global model. Then, the updated global model is once again decomposed into new subnets based on a

new random sampling, and this process repeats until the model converges.

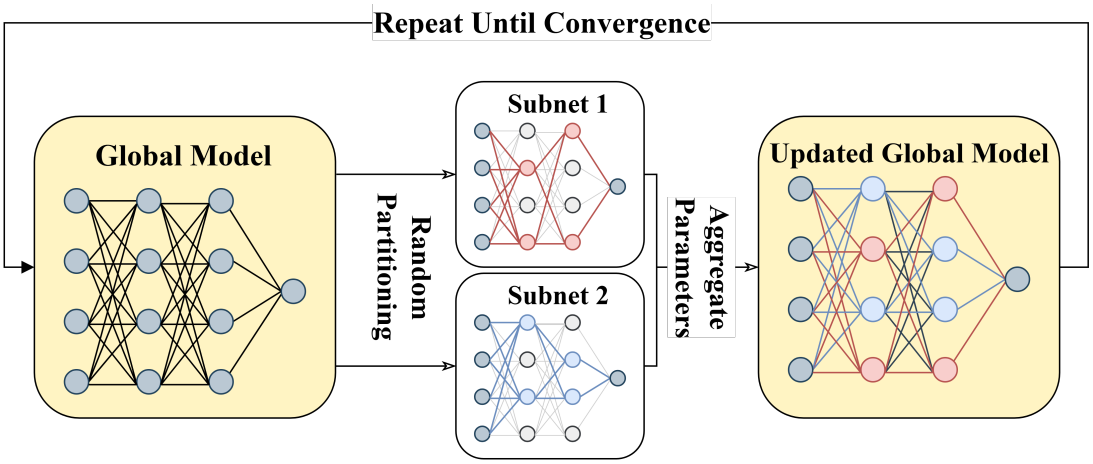


FIGURE 2.2: Overview of a training round in the Independent Subnet Training (IST) approach with two workers. Adopted from [14].

IST has been extended to other architectures as well. The ResIST algorithm [16] adapts IST to ResNet models by assigning different residual blocks, which are less sensitive to pruning [71], to workers and allowing them to train a shallower version of the network. Similarly, the GIST algorithm [72] brings IST to graph neural networks, while AsyncDrop [73] applies the same principles in asynchronous FL settings. IST has also been adapted to hierarchical federated learning: the HIST algorithm [74] assigns a distinct subnet to each edge server. These edge servers then train their assigned subnets using a model-sharing FL approach for a fixed number of local rounds. Then, the results are sent to a central server, which performs global aggregation and distributes updated subnets back to the edge servers for the next round of training.

By avoiding full model replication and frequent gradient exchanges, IST is especially suitable in constrained settings with memory-limited devices or limited network bandwidth. However, as originally proposed, IST inherently assumes

device homogeneity and uniform subnet sizes, which limits its effectiveness in heterogeneous environments, leading to under-utilization, prolonged training time, and increased costs.

Chapter 3

Related Work

In this section, we review the latest work in distributed training and those that strive to tackle the problem of heterogeneity in particular.

3.1 Distributed Machine Learning Training

Many distributed training schemes have been proposed to accelerate the learning of large-scale deep neural networks (DNNs). The most common approach combines data parallelism with synchronous stochastic gradient descent (SGD), where each worker computes gradients on its local mini-batch and synchronizes them at every iteration [75, 76, 77]. To reduce communication overhead, researchers have explored gradient compression [78, 79, 80], hierarchical aggregation [81, 82], and communication-efficient optimization methods [83, 84].

An alternative paradigm is model parallelism, where a single model is partitioned across multiple workers [85, 86, 87]. Pipelining techniques have been introduced to overlap computation and communication [88, 89, 90], and hybrid schemes

combining model and data parallelism have also been explored [91, 92, 93]. Additionally, asynchronous training algorithms such as Hogwild [17], Stale Synchronous Parallel (SSP) [18], and AsynchDrop [73] aim to reduce synchronization overhead by allowing workers to update model parameters at different paces.

3.2 Heterogeneity-Aware Distributed Training

Current techniques and systems for distributed model training often assume that all workers in the cluster are homogeneous, meaning they have similar performance characteristics and resource configurations. However, this assumption does not hold in many real-world scenarios, especially in virtual clusters deployed in data centers or cloud environments [94]. In such settings, worker performance can vary significantly due to several factors, including interference from co-located applications, throttling by the cloud or data center provider, or inherent differences in the hardware configurations of the servers [94]. To address this, recent research has proposed several heterogeneity-aware distributed training strategies.

Chen et al. [95] propose Hare, a non-clairvoyant scheduler that exploits both inter- and intra-job parallelism across heterogeneous GPUs to mitigate straggler effects. While Hare targets workload distribution at the scheduling level, *AdapTrain* focuses on model-level granularity by dynamically adjusting subnet sizes allocated to each worker based on training speed. This allows AdapTrain to synchronize worker progress without external scheduling and to react in real-time to cloud-induced performance fluctuations.

Zhou et al. [96] develop an online scheduler that dynamically assigns jobs to

nodes based on performance modeling, optimizing for job completion latency. Unlike this approach, AdapTrain restructures the model itself during training, enabling fine-grained adaptation to worker speeds. Sun et al. [97] address heterogeneity-induced stragglers through GSSP, which groups workers to balance synchronization. AdapTrain differs by proactively preventing straggler formation through adaptive model partitioning rather than masking their effects post hoc.

Other systems also tackle heterogeneity through adaptive computation placement and parallelization strategies. SOAP [86] integrates both data and model parallelism in heterogeneous settings, while PLink [82] employs topology-aware hierarchical aggregation to mitigate communication bottlenecks. HetPipe [92] combines pipelined model parallelism with data parallelism and introduces Wave Synchronous Parallel to accommodate hardware heterogeneity. Similarly, PipeFisher [98] leverages pipelining and Fisher information to improve GPU utilization in large language model training.

BytePS [99] proposes a summation service architecture where gradient aggregation occurs on underutilized CPU nodes, freeing GPU resources for compute-intensive operations. These approaches demonstrate the potential of adaptive workload balancing in heterogeneous environments.

3.3 Independent Subnet Training and Our Contribution

Independent Subnet Training (IST) [14] partitions the model across workers, ensuring each receives an equal number of neurons. While effective in homogeneous

environments, IST suffers in heterogeneous settings, where slower workers become bottlenecks due to frequent synchronization.

To address this, we propose *AdapTrain*, a heterogeneity-aware extension of IST that dynamically adjusts subnet sizes based on observed worker performance. This allows each worker to train submodels proportional to its capacity, ensuring synchronized round completion and minimizing idle time. AdapTrain preserves IST's advantages—low communication, privacy-by-design, and memory efficiency—while extending its applicability to cloud and edge clusters with heterogeneous hardware or fluctuating performance.

Chapter 4

AdapTrain: Adaptive Model Partitioning

As discussed in 3.3, when deployed in heterogeneous environments, IST's uniform partitioning leads to inefficient training dynamics. Workers with slower performance take longer to complete training on their assigned subnets, while faster workers sit idle, waiting for synchronization. This straggler effect results in underutilized computing resources and increased end-to-end training time. To address this, we follow a two-step methodology:

1. **Training Time Profiling:** First, we investigate the relationship between the duration of training rounds and the subnet sizes, by conducting extensive profiling experiments.

2. **Adaptive Partitioning Strategy:** In the second phase, building upon the results from the first phase, we develop an adaptive approach to dynamically adjust the subnet sizes assigned to each worker before the next training

round. The goal is to partition the model into subnets of varying sizes, ensuring that all workers complete training rounds simultaneously, thereby minimizing idle time and accelerating the training process.

4.1 Problem Definition

Notation. Let n denote the total number of workers (i.e., compute nodes) in the training job. Also, $T(R)$ denotes the time it takes to complete R training rounds, excluding the communication time in between rounds. The total training time $T(R)$ is defined as:

$$T(R) = \sum_{j=1}^{R} T_j = \sum_{j=1}^{R} \max_{i=1}^{n} t_{i,j} \tag{4.1}$$

where T_j denotes the entire duration of the j^{th} training round and $t_{i,j}$ represents the duration of the j^{th} training round for $worker_i$. We also define the partitioning distribution vector of the j^{th} training round, $P_j = \langle p_{i,j} \mid \forall\, 1 \leq i \leq n \rangle$, where $p_{i,j}$ represents the fraction of neurons/blocks assigned to $worker_i$ for the j^{th} training round. Additionally, we define a set of functions $\{f_i \mid \forall\, 1 \leq i \leq n\}$ where f_i models the duration of a training round as a function of the fraction of neurons/blocks assigned to $worker_i$, and is defined as:

$$t_{i,j} = f_i(p_{i,j}) \quad \forall\, 1 \leq i \leq n,\ 1 \leq j \leq R \tag{4.2}$$

Goal. Our goal is to minimize $T(R)$, that is to achieve the minimum completion

time throughout R training rounds. In order to express $T(R)$ in terms of $p_{i,j}$, we merge Eq. 4.1 and 4.2 by substituting $t_{i,j}$ with $f_i(p_{i,j})$:

$$T(R) = \sum_{j=1}^{R} \max_{i=1}^{n} f_i(p_{i,j})$$

Since $T(R)$ is a summation of R terms, each independent, minimizing $T(R)$ requires minimizing each term separately:

$$\text{Minimize:} \quad T_j = \max_{i=1}^{n} f_i(p_{i,j}) \quad \forall\ 1 \leq j \leq R$$

Evidently, f_i is a strictly increasing function, which aligns with the expectation that a larger proportion of the global model assigned to a worker generally leads to a longer training duration due to an increased workload. Also, in each training round, the fraction of neurons/blocks partitioned to workers sums to 1, as each neuron/block is assigned to exactly one worker.

$$\sum_{i=1}^{n} p_{i,j} = 1 \quad \forall\ 1 \leq j \leq R \tag{4.3}$$

By leveraging the properties of strictly increasing functions and the constraint declared in Equation 4.3, we theoretically prove that T_j is minimized when all workers simultaneously complete the j^{th} training round; Thus, we can rewrite the goal as:

$$t_{i,j} = t_{k,j} = T_j^* \quad \forall\ 1 \leq i, k \leq n\ ,\ \ 1 \leq j \leq R \tag{4.4}$$

where T_j^* denotes the optimal duration of the j^{th} training round.

4.2 Training Time Profiling

In this section, we analyze the relationship between the duration of a training round and the partitioned subnet size in two types of neural networks.

4.2.1 Fully Connected Neural Networks

In AdapTrain, model partitioning is performed at the neuron level: each worker is assigned a fraction of the neurons in each hidden layer. However, training time is determined by the number of parameters, not the number of neurons. In fully connected layers, each parameter connects a neuron in one layer to a neuron in the next layer, meaning the number of parameters scales quadratically with the number of assigned neurons. For example, if a worker receives half the neurons in two consecutive layers, it will process one-quarter of the parameters connecting them.

We assume that the duration of an iteration in a fully connected neural network scales linearly with the number of model parameters. This assumption is supported by empirical findings in the context of deep neural networks, where training time is often observed to exhibit linear scaling with model size under idealized conditions, as highlighted in prior work on neural network scaling laws [100].

Based on this assumption, since every training round consists of a fixed number of iterations, the duration of a training round also scales linearly with the number of parameters and hence quadratically with the fraction of neurons in the subnet.

Let L be the number of hidden layers in the model, and S_i denote the number of neurons in each hidden layer. Also, let S_{in}, S_{out} denote the number of neurons in the input and output layers, respectively. $W_{i,j}$, which denotes the number of parameters to be learned by $worker_i$ in the j^{th} training round is defined as follow:

$$\begin{aligned} W_{i,j} &= S_{in}.p_{i,j}S_1 + p_{i,j}S_L.S_{out} + \sum_{l=1}^{L-1} {p^2}_{i,j}S_lS_{l+1} \\ &= p_{i,j}.(S_{in}S_1 + S_L.S_{out}) + {p^2}_{i,j}.\sum_{l=1}^{L-1} S_lS_{l+1} \end{aligned} \tag{4.5}$$

Incorporating the established linear relationship between the number of parameters and training round duration into Equation 4.5, we can express $t_{i,j}$ in terms of $p_{i,j}$:

$$\begin{aligned} t_{i,j} \propto W_{i,j} \Rightarrow t_{i,j} = \alpha_i \left(\sum_{l=1}^{L-1} S_lS_{l+1}\right) {p^2}_{i,j} + \\ \alpha_i(S_{in}S_1 + S_L.S_{out})p_{i,j} \end{aligned} \tag{4.6}$$

The first term captures the quadratic dependency on $p_{i,j}$, associated with the sizes of consecutive hidden layers, while the second term represents the linear dependency appearing from connections to the input and output layers. In typical fully connected neural networks, the hidden layers often dominate in size compared to the input and output layers. For example, in image classification tasks, S_{in} is usually the number of pixels in the input image, and S_{out} is the number of classes, both of which are orders of magnitude smaller than the sizes of hidden layers $S_1, \ldots, S_L$. Therefore, the contribution of the linear term to the training round

duration is negligible compared to the quadratic term. By omitting the linear term, we simplify the model to focus on the dominant factor:

$$\begin{aligned} t_{i,j} &\approx \alpha_i \left(\sum_{l=1}^{L-1} S_l S_{l+1} \right) {p^2}_{i,j} \\ \Rightarrow t_{i,j} &= C_i {p^2}_{i,j} \end{aligned} \tag{4.7}$$

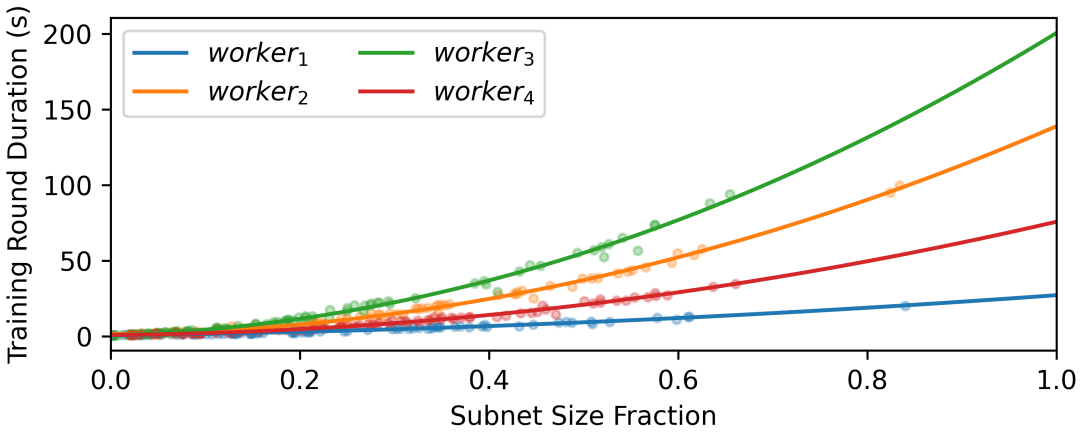


FIGURE 4.1: Duration of a single training round vs. the fraction of model neurons assigned to each worker in a fully connected neural network. Each curve represents a different worker, illustrating how training round duration increases quadratically with subnet size.

This simplification reduces computational complexity and accurately represents the training duration, as the omitted term has minimal impact. As a result, we derive that the training round duration for a worker increases quadratically as the fraction of neurons assigned to it grows. As shown in Fig. 4.1, our profiling experiments support this result. Therefore, the duration of a training round for each worker can be expressed as a function of the fraction of the global model's neurons assigned to each worker during that round:

$$\frac{t_{i,j}}{{p^2}_{i,j}} = \frac{t_{i,j+1}}{{p^2}_{i,j+1}} \quad \forall\, 1 \leq i \leq n\,,\ 1 \leq j < R \tag{4.8}$$

4.2.2 Residual Networks

The duration of an iteration in a ResNet model is expected to increase linearly with the number of residual blocks, as each block involves specific computations that cumulatively determine the overall computation time. Residual blocks typically include multiple convolutional layers, batch normalization, and activation functions, all adding a fixed computational overhead per block [101].

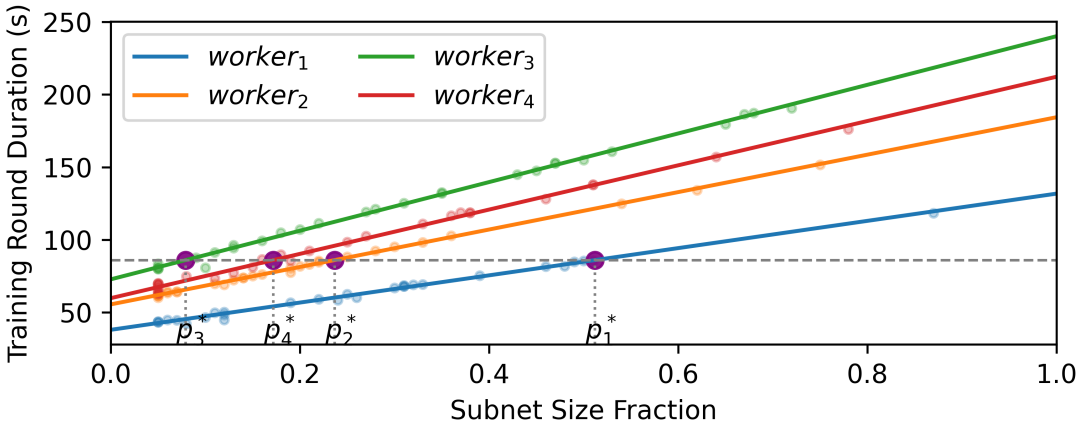


FIGURE 4.2: Duration of a single training round vs. the fraction of partitionable residual blocks assigned to each worker in ResNet-101. Each curve represents a different worker, illustrating how training round duration scales linearly with the number of residual blocks. $P = \langle {p^*}_1, \ldots, {p^*}_4 \rangle$ is the optimal partitioning distribution vector used in the ideal initialization baseline throughout the training.

Fig. 4.2 illustrates the linear relationship between the duration of training rounds and the fraction of residual blocks from the global ResNet model assigned to workers for training. The positive y-intercept for workers in Fig. 4.2 is due to

the fact that only the blocks in the third section of the model are considered to be partitioned among workers in independent subnet training while other blocks are shared across all workers [16]. So, the y-intercept demonstrates the training time of the other sections of the model.

Therefore, for a ResNet model, we can express $t_{i,j}$ as:

$$t_{i,j} = f(p_{i,j}) = a_i p_{i,j} + b_i \quad \forall\, 1 \leq i \leq n \tag{4.9}$$

Since the blocks within the third section of the model account for the majority of the layers, their contribution to training time is much more significant than that of other sections, which are shared between all workers. Furthermore, as the model becomes deeper, the ratio of total blocks considered for partitioning becomes even larger, as deeper ResNet variants only add blocks to the third section [16]. Drawing upon these results, the contribution of the constant term in Eq. 4.9 to the training round duration is negligible compared to the linear term, especially for deeper ResNet models. By omitting the constant term, we simplify the model to focus on the dominant factor:

$$\begin{aligned} t_{i,j} = f(p_{i,j}) \approx a_i p_{i,j} \quad & \forall\, 1 \leq i \leq n \\ \Rightarrow \frac{t_{i,j}}{p_{i,j}} = \frac{t_{i,j+1}}{p_{i,j+1}} \quad & \forall\, 1 \leq i \leq n\,,\ 1 \leq j < R \end{aligned} \tag{4.10}$$

4.3 Adaptive Partitioning Strategy

To address the problem defined in Section 4.1, we employ adaptive training. The primary reason for using this strategy is the inherent variability in computational resources on the workers in the cloud, which can result in inefficiencies during training. A static approach that partitions the model uniformly fails to account for differences in processing speeds, causing faster workers to wait idly while slower ones finish each training round. This idle time leads to the underutilization of more capable workers and increases training costs. An additional benefit of AdapTrain is that it operates without requiring prior knowledge of the workers' computational capacities. While traditional methods often rely on predefined resource estimates to allocate tasks effectively, AdapTrain instead learns from each training round, adjusting partition sizes based on observed performance, thereby eliminating the need for resource profiling upfront. This makes it cloud-agnostic and highly practical for heterogeneous environments where worker capacities may not be well-defined or may change over time. Another reason for adopting an adaptive strategy is the unpredictable nature of cloud environments, where workloads can fluctuate unexpectedly. Static partitioning strategies are less effective under these conditions, as they lack the flexibility to respond to sudden changes in resource availability or performance. AdapTrain reallocates workload distribution on the fly using an adaptive approach, assigning smaller partitions to heavily loaded nodes and larger partitions to under-utilized ones. This responsiveness helps maintain balanced training times across all workers, even in dynamic environments, improving overall efficiency, scalability and robustness.

Another key reason for using an adaptive approach is that our optimization

objective inherently motivates the use of such a strategy. Eq. 4.4 indicates that the duration of each training round should be minimized independently to minimize the end-to-end training time. Furthermore, Eq. 4.8 suggests that the partitioning distribution among workers must adapt based on their performance in the previous round.

4.3.1 Adaptive Partitioning in FCNNs

At the beginning of the training, AdapTrain uses a uniform partitioning distribution, assigning subnets of equal size to each worker. At the end of each training round, AdapTrain measures the completion time of each worker and calculates the best possible partitioning distribution to achieve simultaneous completion of the subsequent training round among workers. Deriving $t_{i,j+1}$ from Eq. 4.4, we can rewrite Eq. 4.8:

$$\begin{aligned} &\frac{t_{i,j}}{{p^2}_{i,j}} = \frac{t_{i,j+1}}{{p^2}_{i,j+1}} = \frac{T^*_{j+1}}{{p^2}_{i,j+1}} \qquad \forall \quad 1 \leq i \leq n \\ &\Rightarrow \quad p_{i,j+1} = \sqrt{\frac{T^*_{j+1}.{p^2}_{i,j}}{t_{i,j}}} = \sqrt{\frac{T^*_{j+1}}{t_{i,j}}}.p_{i,j} \end{aligned} \tag{4.11}$$

Then, considering Eqs. 4.3 and 4.11, we have:

$$\begin{aligned} &\sum_{i=1}^{n} p_{i,j+1} = 1 \quad \Rightarrow \quad \sum_{i=1}^{n} \sqrt{\frac{{T^*}_{j+1}}{t_{i,j}}}.p_{i,j} = 1 \\ &\Rightarrow \quad \sqrt{{T^*}_{j+1}} \sum_{i=1}^{n} \frac{p_{i,j}}{\sqrt{t_{i,j}}} = 1 \\ &\Rightarrow \quad {T^*}_{j+1} = \frac{1}{\left(\sum_{i=1}^{n} \frac{p_{i,j}}{\sqrt{t_{i,j}}}\right)^2} \end{aligned} \tag{4.12}$$

Finally, we substitute ${T^*}_{j+1}$ in Eq. 4.11 with the result derived from Eq. 4.12:

$$p_{i,j+1} = \frac{p_{i,j}}{\sqrt{t_{i,j}\left(\sum_{k=1}^{n}\sqrt{\frac{{p^2}_{k,j}}{t_{k,j}}}\right)^2}} \tag{4.13}$$

So, at the end of the j^{th} training round, AdapTrain collects workers' training round durations, $t_{i,j}$, and their fraction of neurons from the global model, $p_{i,j}$. It then calculates the partitioning distribution vector $P_{j+1} = \langle p_{i,j+1} \mid \forall\, 1 \leq i \leq n \rangle$ for the following training round using the result from Eq. 4.13. Then, the workers are synchronized, and the model is repartitioned into new subnets based on the new partitioning distribution before the next training round begins. AdapTrain repeats this process after each training round until convergence or the desired level of accuracy is reached.

4.3.2 Adaptive Partitioning in ResNets

AdapTrain begins with a uniform partitioning distribution of residual blocks within the third section of the model, forming subnets of equal size and depth. At the end of each training round, AdapTrain collects the completion time of that round from each worker and calculates the optimal partitioning distribution for the next round by incorporating Eqs. 4.3, 4.4 and 4.10:

$$\begin{aligned} \sum_{i=1}^{n} p_{i,j+1} = 1 \;\Rightarrow\; T^*_{j+1} \sum_{i=1}^{n} \frac{p_{i,j}}{t_{i,j}} = 1 \\ \Rightarrow T^*_{j+1} = \frac{1}{\sum_{i=1}^{n} \frac{p_{i,j}}{t_{i,j}}} \end{aligned} \tag{4.14}$$

Finally, we substitute T^*_{j+1} in Eq. 4.14 with the result derived from Eq. 4.10:

$$p_{i,j+1} = T^*_{j+1} \frac{p_{i,j}}{t_{i,j}} = \frac{p_{i,j}}{t_{i,j} \sum_{k=1}^{n} \frac{p_{k,j}}{t_{k,j}}} \tag{4.15}$$

Chapter 5

Experimental Evaluation

In this section, we report on our extensive experiments to evaluate AdapTrain when applied to distributed training on public cloud infrastructures, which are inherently heterogeneous and changing environments.

5.1 Experiments Setup

We conduct experiments on real-world cloud resources to evaluate the effectiveness of AdapTrain. Our cluster consists of four Kubernetes-managed VMs provisioned on Compute Canada's Arbutus Cloud [108]. VMs have different CPU

Table 5.1: Workers' Computational Resources

Node	GPU	CPU Model	CPUs	RAM (GB)
$worker_1$	V100 [102]	Xeon (Skylake) [103]	16	24
$worker_2$	T4 (Simulated) [104]	Xeon (Skylake)	2	15
$worker_3$	A2 (Simulated) [105]	Xeon (Skylake)	1	7.5
$worker_4$	P4 (Simulated) [106]	Core (Broadwell) [107]	4	30

configurations and memory capacities but identical GPU models (detailed in Table 5.1). To simulate the GPU heterogeneity common in public cloud settings, controlled delays were forced during training to emulate the behavior of lower-end GPUs. The Nvidia V100 served as the reference device. For each emulated GPU, a slowdown ratio was estimated based on publicly reported FP32 throughput [109, 110, 111, 112]. After every training step, a delay proportional to the inverse of the target GPU's relative speed was applied. This approach preserves the model's computation graph and communication pattern while introducing realistic differences in wall-clock time across workers. All experiments were repeated with three random seeds and averaged for stability. We evaluate AdapTrain on two representative deep-learning models:

- A 3-layer fully connected neural network (FCNN) trained on the Google Speech Commands dataset [113] for audio classification (35 classes).
- A ResNet-101 model trained on CIFAR-100 [114] for image classification.

These choices reflect the current applicability of the IST framework rather than a fundamental limitation of AdapTrain. As IST continues to expand to new architectures, AdapTrain can be readily extended as well.

5.2 Baselines

To evaluate AdapTrain, we compare its performance to several baselines:
Standard IST/ResIST: We use the standard IST/ResIST, which is heterogeneity-agnostic, as a baseline to evaluate our approach. In terms of performance, this

baseline represents a lower bound to our work. We use the same IST/ResIST implementation as in [14, 16] without modifications.

Distributed Data-Parallel: We used PyTorch's DDP module with gradient accumulation and heterogeneity-aware data partitioning [115, 116]. Based on an initial profiling phase, we assign data shards to workers proportional to their relative throughput to ensure fairer comparisons in heterogeneous environments. Specifically, if a worker's batch completion time was t_i, its data share d_i was scaled inversely proportional to t_i. This enables more balanced training steps under hardware diversity.

Ideal Initialization: We introduce the ideal initialization scenario, a static approach that represents an ideal upper bound for our work, in which the partitioning distribution is ideally configured from the beginning of training and remains unchanged throughout training (i.e., perfect knowledge is assumed in advance). In other words, this approach accurately accounts for system heterogeneity but fails to adapt to the evolving conditions during the training process. The ideal partitioning distribution is learned by profiling the training round duration of workers upfront in an ideal environment with minimal resource/workload fluctuations. While achieving such initialization is computationally expensive in practice and requires costly profiling upfront, we do not factor in the cost of this profiling when presenting our results. Since ideal initialization is a static approach, the partitioning distribution doesn't change over the course of training. Let $P = \langle p_i \mid \forall\ 1 \leq i \leq n \rangle$ denote the static partitioning distribution throughout training, $P^* = \langle p_i^* \mid \forall\ 1 \leq i \leq n \rangle$ denote the static ideal partitioning distribution and t_i denote the time taken by $worker_i$ to complete a training round being

assigned a fraction p_i of the global model, and T^* denote the ideal simultaneous completion time of all workers in all training rounds. As illustrated in Fig. 4.1,4.2, t_i can be expressed as a function of p_i, that is $f_i(p_i) = t_i$. We first conduct a profiling phase by allowing the workers to train for a few rounds using various partitioning distributions, and then we apply a curve fit to achieve an approximation of f_i. Also, as per the normalization equation (i.e., Eq. 4.3), the p_i values sum to one for each training round. Furthermore, as derived in Eq. 4.4, we ideally want all workers to complete training rounds simultaneously.

$$\begin{cases} f_i(p_i^*) = f_k(p_k^*) = T^* & \forall\ 1 \leq i, k \leq n \\ \sum_{i=1}^{n} p_i^* = 1 \end{cases} \tag{5.1}$$

In order to achieve the ideal partitioning distribution, we solve Eq. 5.1 for p_i^*. We first prove the system has a solution. Since each f_i is a strictly increasing continuous function, it would be one-to-one (i.e. invertible) and onto (surjective), and its inverse function, f_i^{-1}, would also be one-to-one and surjective. Let $F(t) = \sum_{i=1}^{n} f_i^{-1}(t)$. Again, since each f_i^{-1} is a one-to-one and onto function, their summation, F, would also be a one-to-one and onto function. Therefore:

$$\forall\ x \in \mathbb{R}^+ \quad \exists\ t \ \text{ such that } F(t) = x$$

$$\text{put } x = 1 \Rightarrow \exists\ t \text{ such that } F(t) = 1$$

$$\Rightarrow p_i^* = f_i^{-1}(t) = f_i^{-1}(F^{-1}(1))$$

Since solving this equation is not analytically feasible, we can reformulate it as an optimization problem. The objective is to find p_i values that minimize the

objective function, which is $\max_{i=1}^{n} f(p_i)$ while satisfying the constraints $\sum_{i=1}^{n} p_i = 1$, and $p_i \in (0, 1)$. Once we find the optimal p_i values, we train the model using the derived ideal partitioning distribution P^* throughout the whole training process.

5.3 Performance Metrics

End-to-End Training Time: The key performance metric for evaluating our adaptive approach is the *end-to-end training time*. This metric captures the total training duration, from the initial round to the point at which a predetermined accuracy level is reached. Notably, the end-to-end training time excludes testing times, focusing on the efficiency of the training rounds.

Non-Idle Ratio: Another key metric is what we call non-idle ratio, which reflects the proportion of time a worker spends actively training (as opposed to waiting for synchronization). This metric helps capture inefficiencies introduced by stragglers, particularly in heterogeneous environments where faster workers are often underutilized due to synchronization barriers. The impact of idle time is especially pronounced for high-performance nodes, which typically incur higher operational costs (e.g., compute, memory, and energy consumption). Consequently, idle time among faster workers can increase overall training costs and prolong training time, as these resources are not fully utilized. While this metric does not account for hardware-level efficiency (e.g., achieved FLOPS), it is designed to capture synchronization-induced idle time and provides insight into resource underutilization due to workload imbalance.

$$\text{Non-Idle Ratio for } worker_i = \frac{\sum_{j=1}^{R} t_{i,j}}{\sum_{j=1}^{R} \max_{k=1}^{n} t_{k,j}}$$

GPU Utilization: We report GPU Streaming Multiprocessor (SM) utilization to provide additional insight into how effectively each method uses the available GPU resources during training. This metric reflects the average percentage of time the GPU's compute units are active, offering a hardware-level perspective on system load. While SM utilization does not directly measure training efficiency, it helps investigate differences in hardware usage patterns among baselines.

Final Accuracy: Lastly, we measure the maximum accuracy achieved by the model after 100 epochs of training, and report it as the final accuracy. This metric ensures that efficiency improvements do not compromise model performance, allowing us to verify that our adaptive strategy maintains or improves upon the accuracy of baseline methods.

5.4 Results and Analysis

In this section, we evaluate AdapTrain's performance against the baselines based on the following criteria: To ensure fair comparisons, all experiments were conducted under identical conditions, including the same simulated fluctuations in computational resources. These fluctuations were designed to mimic real-world scenarios in a public cloud environment and were consistently applied across all experiments.

5.4.1 Workload Balancing

The primary objective of AdapTrain is to distribute the workload evenly among workers, enabling them to complete training rounds simultaneously. Failing to achieve this objective would result in significant periods of idle time for faster, more capable workers. To evaluate the effectiveness of AdapTrain in balancing the workloads across heterogeneous workers, we examine three key metrics plotted throughout training: (1) the distribution of training round completion times across workers, (2) the training time of the slowest worker, (3) the mean non-idle ratio across workers. Each plot provides insight into how AdapTrain reduces workload disparities and optimizes resource usage throughout training.

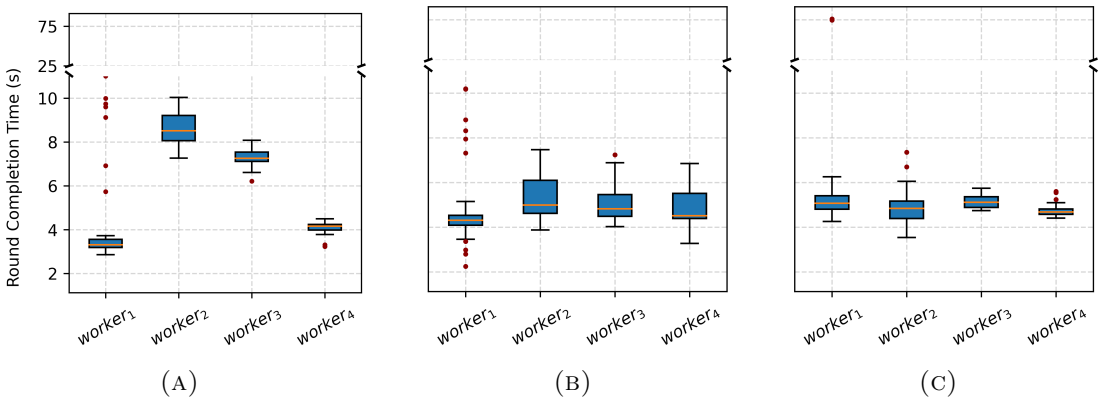
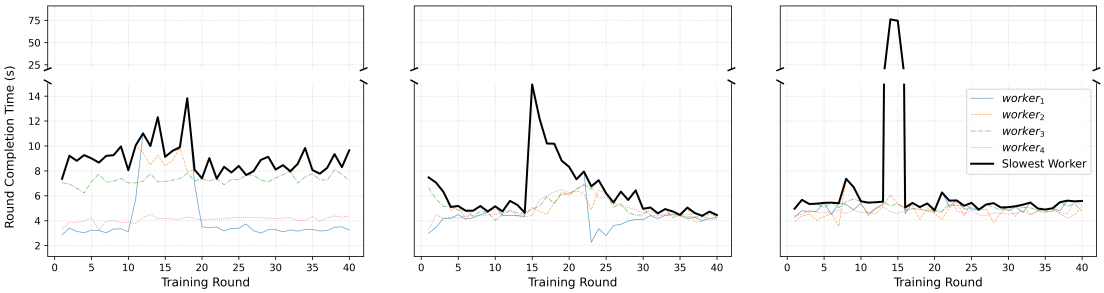


FIGURE 5.1: Distribution of training round completion times for each worker throughout training a 3-layer FCNN on public cloud environment using (A) IST (B) AdapTrain (C) Ideal-Initialization. The broken y-axis helps illustrate both the main trend of round completion times throughout training as well as outliers.

Fig. 5.1 presents the distribution of training round completion times across workers for IST, AdapTrain, and Ideal Initialization. While IST suffers from severe straggler effects due to uniform partitioning, both AdapTrain and Ideal-Init attempt to mitigate this by assigning larger subnets to faster workers. However,

a key distinction lies in their response to runtime fluctuations. Although workers 2–4 under Ideal-Init show slightly lower median round durations than in AdapTrain, the overall training time is dictated by the slowest worker in each round due to the synchronized nature of IST. In this case, worker 1 (the slowest) frequently experiences prolonged round durations in Ideal-Init, as the static partitioning fails to adapt to real-time fluctuations. This results in significantly higher outliers for worker 1 in Fig.5.1(c), as highlighted by the broken y-axis. In contrast, AdapTrain (Fig.5.1(b)) adapts its partitioning each round, effectively reducing the tail latency caused by such stragglers. Consequently, AdapTrain demonstrates superior robustness and efficiency, particularly in environments with inherent heterogeneity and fluctuations, such as the public cloud.



(A) (B) (C)

FIGURE 5.2: Training rounds durations across workers throughout training a 3-layer FCNN on public cloud environment using (A) IST (B) AdapTrain (C) Ideal-Initialization. The broken y-axis helps illustrate the main trend of round completion times throughout training and sudden, significant spikes. The black curve indicates the last worker to complete training at each training round.

Fig. 5.2 compares AdapTrain to the baselines (IST and ideal initialization) regarding workload balancing and robustness by illustrating each worker's time to complete each training round throughout the training process. AdapTrain begins with a uniform partitioning distribution among workers, similar to IST. Therefore,

there is initially a notable gap between the slowest worker and the others in terms of training round durations. As training progresses, our adaptive approach gradually modifies the partitioning distribution according to each worker's performance in prior training rounds, reducing the gap. After a few rounds, AdapTrain nearly converges to the optimal partitioning distribution, resembling the behaviour of the ideal initialization baseline. Then, around the 15^{th} training round, a sudden spike occurs caused by the emulated public cloud fluctuations, which impacts $worker_1$. AdapTrain immediately reacts to this incident by adjusting the partitioning distribution, ensuring minimal disruption and continued progress. In contrast, the ideal initialization baseline fails to adapt to the change, leading to a significant spike in training round completion times during the fluctuation period. This responsiveness of AdapTrain allows it to complete approximately 5x more training rounds than the ideal initialization baseline over the same period.

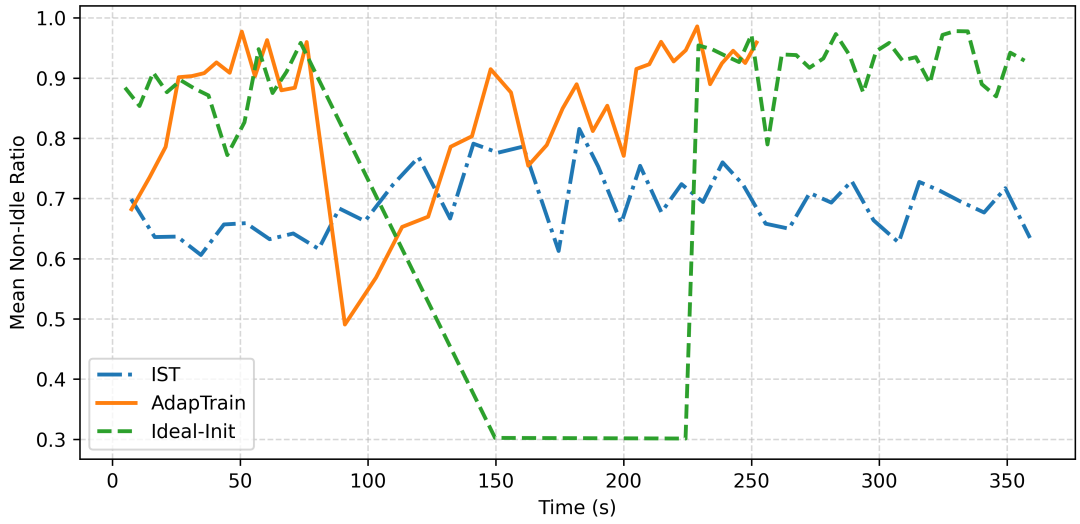


FIGURE 5.3: Mean non-idle ratio across workers throughout training a 3-layer FCNN on a public cloud environment.

Fig. 5.3 captures the mean non-idle ratio across workers over training time.

AdapTrain begins with a comparatively low mean non-idle ratio, similar to IST, due to the initial uniform partitioning distribution and gradually converges to the maximum non-idle ratio, like Ideal-Init, before a sudden fluctuation emerges at around $t = 80s$. Again, AdapTrain quickly responds to this occurrence by adjusting the partitioning distribution among workers and slowly reaches higher utilization, while the Ideal-Init approach struggles. To further complement Fig. 5.3, we present Table 5.2, which provides detailed information on Fig. 5.3, AdapTrain achieves 82.1% non-idle ratio, which is significantly higher compared to 71.4% of Ideal-Init and 66.3% of IST.

TABLE 5.2: Workers' non-idle ratio throughout training a 3-layer FCNN on Google Speech dataset for 100 training rounds

	IST				AdapTrain				Ideal-Init			
Total Training Time (s)	940				605				768			
Worker	w_1	w_2	w_3	w_4	w_1	w_2	w_3	w_4	w_1	w_2	w_3	w_4
Workers Training Time (s)	405	890	723	476	503	496	488	500	647	475	501	569
Workers Non-Idle Ratio	43%	95%	73%	51%	83%	82%	81%	83%	84%	62%	65%	74%
Overall Non-Idle Ratio	66.3%				**82.1%**				71.4%			

Table 5.3 presents the average GPU SM utilization of each worker, along with the overall average and standard deviation, during ResNet-101 training on CIFAR-100. While GPU utilization can be influenced by factors beyond workload size (e.g., kernel efficiency or memory access patterns), it could provide a useful approximation of per-worker resource usage in practice. As expected, the DDP baseline achieves the highest overall utilization (77.8%), but at the cost of requiring full model replication and high communication overhead. IST, while communication-efficient, suffers from highly imbalanced GPU usage, particularly with $worker_3$ reaching 75.4% while others remain significantly underutilized—resulting in the lowest overall average (49.6%) and highest variability (15.67 percentage points).

AdapTrain improves upon this by balancing the workload more evenly across workers, narrowing the gap in individual utilizations, and reducing the standard deviation to just 5.62 percentage points. Although its average utilization (67.2%) is lower than DDP, it reflects a more balanced and stable distribution.

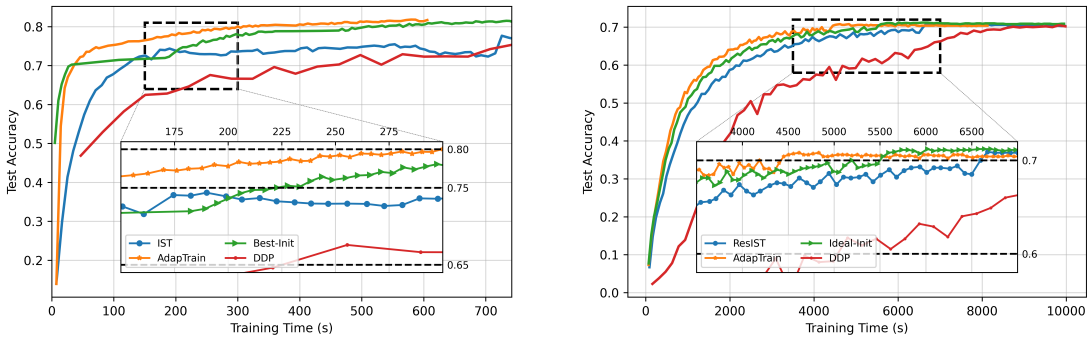
TABLE 5.3: Average GPU SM Utilization per worker across different methods throughout training ResNet-101 on CIFAR-100.

Method	Worker 1	Worker 2	Worker 3	Worker 4	Overall Avg	STD
DDP	55.6%	80.9%	89.6%	85.1%	77.8%	13.1
IST	33.9%	41.4%	75.4%	47.9%	49.6%	15.67
AdapTrain	60.3%	69.2%	75.2%	63.9%	67.2%	5.62

5.4.2 Convergence Speed

A key evaluation point of our adaptive approach is its effect on convergence speed, remarkably, on how quickly our method reaches a target test accuracy compared to baseline methods. To assess this, we present Fig. 5.4 illustrating test accuracy over time for our adaptive strategy against baselines. Fig. 5.4(a) compares the training time of the 3-layer fully connected neural network for AdapTrain against the baseline methods, and Fig. 5.4(b) depicts that of the ResNet-101 model.

One promising observation from Fig. 5.4 is how AdapTrain initially aligns with IST and gradually surpasses IST and other baselines, ensuring the optimal end-to-end training time. We also present Table 5.4 that summarizes the time taken by each method to converge. The table shows that in the FCNN experiment, AdapTrain achieves up to a 2.5x, 5.7x, and 8.2x speedup in training time compared to Ideal-Init, IST, and DDP, respectively. Notably, DDP and IST failed to achieve 80% accuracy level in 100 epochs of training. For the ResNet experiment,



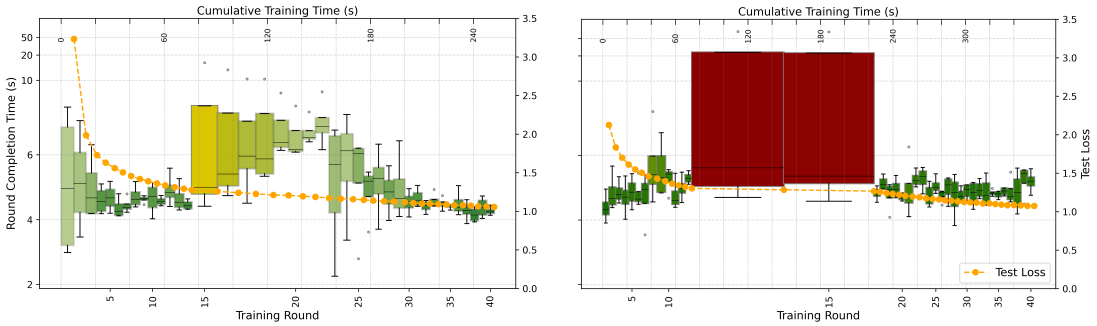
(A) (B)

FIGURE 5.4: Comparison of test accuracy over training time for: (A) 3-Layer FCNN and (B) ResNet-101

at 70% accuracy, AdapTrain exhibits a reduction of approximately 37% in training time compared to IST, 62% compared to DDP, and 19% compared to Ideal Initialization.

To further evaluate the effectiveness of our adaptive approach in terms of convergence speed, we present Fig. 5.5 that compares AdapTrain to Ideal-Init by illustrating the progression of the loss function as well as the distribution of training round durations across workers throughout training. This figure represents how AdapTrain accelerates training compared to the ideal initialization method, which is the best possible non-adaptive approach. While under the same workload/resource fluctuations, AdapTrain completes around 10x more training rounds from $t = 80s$ to t=$220s$ than Ideal-Init, leading to a significant decrease in the test loss.

Table 5.4 shows that AdapTrain consistently reaches target accuracy levels faster than all baseline methods across both experimental settings. Compared to the standard IST, AdapTrain reduces the time to reach mid-level accuracy by up



(A) AdapTrain (B) Ideal Initialization

FIGURE 5.5: Distribution of training round completion times across workers during the first 40 training rounds for (A) AdapTrain and (B) Ideal-Initialization approach, both training a 3-layer fully connected neural network on Google Speech Commands. The top axis illustrates the cumulative training time, while the bottom axis shows the training round. The left y-axis uses a piecewise linear scale to display the main trend and occasional spikes, while the right y-axis represents the test loss. Each box plot's width represents the round duration, emphasizing the variability across rounds, and the colour gradient (green to red) visually encodes the round duration, with green indicating shorter rounds and red representing longer ones.

to 82%, and outperforms DDP by as much as 87%. Even when compared to the Ideal-Init, AdapTrain achieves speedups of up to 41%.

TABLE 5.4: End-to-end training time (in seconds) to reach various levels of test accuracy for a 3-layer FCNN on the Google Speech dataset and ResNet-101 on CIFAR-100.

	Training Time (s)			
	IST	DDP	Ideal-Init	AdapTrain
Accuracy (%)	3-Layer FCNN			
65	88	255	**15**	25
75	520	741	222	**90**
80	–	–	511	**299**
Accuracy (%)	ResNet-101			
60	2322	6432	1827	**1594**
70	6595	10924	5194	**4186**

5.4.3 Final Accuracy

An additional key metric for assessing our adaptive method is the final accuracy achieved by the model compared to the baselines. To illustrate this, we include Table 5.5, which shows the final accuracy for each approach, demonstrating how our adaptive strategy maintains model performance. While not explicitly optimized for convergence, AdapTrain maintained competitive final accuracy. However, we note that dynamic partitioning may introduce variance in training dynamics, requiring occasional tuning.

Table 5.5: Final test accuracy achieved by each method.

	Final Accuracy (%)	
	3-Layer FCNN	ResNet-101
IST	79.8	70.8
DDP	79.4	71.0
Ideal-Init	81.5	**71.2**
AdapTrain	**81.8**	70.9

Chapter 6

Conclusion

This chapter concludes the thesis by summarizing the main contributions and findings, followed by a discussion of current limitations and directions for future work. The research focused on improving the efficiency of distributed training under heterogeneous conditions by extending the Independent Subnet Training (IST) framework with an adaptive partitioning strategy.

6.1 Summary

This thesis introduced *AdapTrain*, a runtime-adaptive training framework that addresses the inefficiencies of Independent Subnet Training (IST) in heterogeneous environments by developing an adaptive approach that dynamically allocates subnet sizes based on each worker's capacity. The key contributions and findings are summarized below:

- **IST's Limitation:** We highlight a fundamental limitation of Independent Subnet Training (IST) in heterogeneous environments, showing how uniform

model partitioning leads to worker idle time and inefficient resource usage.

- **Adaptive Model Partitioning:** We introduced a feedback-driven mechanism that dynamically assigns larger subnets to faster workers and smaller subnets to slower ones, ensuring synchronized training completion across the cluster.

- **AdapTrain:** We develop AdapTrain as a lightweight and modular extension of the IST framework, requiring minimal coordination overhead and preserving IST's core advantages. The full implementation of AdapTrain, including the Autonomic Management System (AMS), representative models, configuration files and documentation that are required to replicate this study, are publicly available in the PACS Lab GitHub repository.

- **Idle Time:** AdapTrain reduces the idle time by balancing the workload according to actual performance rather than static assumptions, leading to better efficiency in heterogeneous environments.

- **Convergence Guarantee:** IST's guaranteed convergence has been established under certain assumptions, including a uniform distribution of neurons or blocks across workers. A key aspect of IST is the scaling applied to ensure convergence and a sampling approach to mitigate the accumulation of random effects (i.e., distributional shift). We adapted the scaling aspects to account for non-uniformity. The sampling approach and techniques for correcting distributional shifts remain largely similar to those of the original IST.

- **Training Speedup:** AdapTrain demonstrated significant speedups in two settings: a 3-layer fully connected network and ResNet-101, showing up to 75% faster convergence compared to the standard IST, substantial improvement over data-parallel training without compromising accuracy, and near-optimal performance compared to an ideal workload initialization baseline.

- **Maintained Accuracy:** Despite dynamic partitioning, AdapTrain preserved final model performance across all experiments, confirming the reliability of the approach.

6.2 Discussion, Limitations, and Future Work

While AdapTrain achieves strong results, several limitations and areas for future improvement remain:

Architecture Support: It is essential to note that AdapTrain has been conceptualized using a bottom-up approach to evolve into a model-agnostic, autonomic system capable of adaptively optimizing overall training time. We started the design with a distributed training system based on IST/ResIST. More specifically, independent subnet training, and consequently AdapTrain, is inherently designed for fully connected neural networks and residual networks. This restricts the broader applicability of AdapTrain to modern architectures such as Transformers, in which attention layers account for 40-50% of the computation [117, 118]. AdapTrain would only apply to the fully connected layers for such models, limiting its

advantages and impact on the training time. Addressing such limitations is part of our immediate future research.

Experimental Scope: Regarding experimental evaluation, the current work focuses on a fully connected 3-layer neural network and ResNet-101. For a more comprehensive evaluation, future experiments should extend to more complex and deeper model architectures to assess the effectiveness of our adaptive approach in high-performance, resource-intensive training environments. Our initial results indicate that the AdapTrain is more effective for deeper models.

Fault Tolerance: Another important consideration is fault tolerance. In real-world cloud settings, node failures are common during long training jobs. AdapTrain can be extended to handle partial failures by modeling a failed node as one with full utilization, effectively excluding it from future training rounds. Incorporating such resilience into the system will be key to supporting reliable distributed training at scale.

Convergence Control: While AdapTrain dynamically adjusts workloads based on performance feedback, it does not explicitly monitor or optimize for convergence. Although accuracy was preserved in our experiments, certain hyperparameters such as minimum subnet size or repartitioning frequency may require tuning to ensure convergence on other models or datasets. Future iterations of AdapTrain could incorporate convergence-aware adaptations to make the system more robust across tasks.

In conclusion, this thesis contributes a practical and adaptive approach to

model partitioning for distributed training in heterogeneous environments. By improving resource utilization and reducing training time without compromising accuracy, AdapTrain lays the foundation for future systems that are both efficient and resilient, capable of adapting to real-world variability in large-scale machine learning deployments.

Bibliography

[1] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[3] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[4] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[6] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, *et al.*, "Learning transferable visual models from

natural language supervision," in *International conference on machine learning*, pp. 8748–8763, PmLR, 2021.

[7] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International conference on machine learning*, pp. 8821–8831, Pmlr, 2021.

[8] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021.

[9] N. Ahmed and M. Wahed, "The de-democratization of ai: Deep learning and the compute divide in artificial intelligence research," *arXiv preprint arXiv:2010.15581*, 2020.

[10] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai," *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[11] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[12] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, *et al.*, "Laion-5b: An open large-scale dataset for training next generation image-text models," *Advances in neural information processing systems*, vol. 35, pp. 25278–25294, 2022.

[13] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 2262–2270, 2021.

[14] B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. Jermaine, "Distributed learning of fully connected neural networks using independent subnet training," *Proceedings of the VLDB Endowment*, 2022.

[15] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018.

[16] C. Dun, C. R. Wolfe, C. M. Jermaine, and A. Kyrillidis, "Resist: Layer-wise decomposition of resnets for distributed training," in *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence* (J. Cussens and K. Zhang, eds.), vol. 180 of *Proceedings of Machine Learning Research*, pp. 610–620, PMLR, 01–05 Aug 2022.

[17] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.

[18] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *Advances in neural information processing systems*, vol. 26, 2013.

[19] B. Radovič, M. Canini, and V. Pejović, "Review and comparative evaluation of resource-adaptive collaborative training for heterogeneous edge devices," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 10, no. 1, pp. 1–35, 2025.

[20] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[21] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine learning and systems*, vol. 2, pp. 429–450, 2020.

[22] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang, "On the convergence of fedavg on non-iid data," *arXiv preprint arXiv:1907.02189*, 2019.

[23] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečnỳ, S. Kumar, and H. B. McMahan, "Adaptive federated optimization," *arXiv preprint arXiv:2003.00295*, 2020.

[24] A. Yousefpour, S. Guo, A. Shenoy, S. Ghosh, P. Stock, K. Maeng, S.-W. Krüger, M. Rabbat, C.-J. Wu, and I. Mironov, "Green federated learning," *arXiv preprint arXiv:2303.14604*, 2023.

[25] M. T. Munir, M. M. Saeed, M. Ali, Z. A. Qazi, A. A. Raza, and I. A. Qazi, "Learning fast and slow: Towards inclusive federated learning," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 384–401, Springer, 2023.

[26] A. M. Abdelmoniem, A. N. Sahu, M. Canini, and S. A. Fahmy, "Refl: Resource-efficient federated learning," in *Proceedings of the eighteenth European conference on computer systems*, pp. 215–232, 2023.

[27] A. M. Abdelmoniem, C.-Y. Ho, P. Papageorgiou, and M. Canini, "Empirical analysis of federated learning in heterogeneous environments," in *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pp. 1–9, 2022.

[28] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, *et al.*, "Advances and open

problems in federated learning," *Foundations and trends® in machine learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[29] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan, *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[30] H. Chang, V. Shejwalkar, R. Shokri, and A. Houmansadr, "Cronus: Robust and heterogeneous collaborative learning with black-box knowledge transfer," *arXiv preprint arXiv:1912.11279*, 2019.

[31] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[32] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 535–541, 2006.

[33] D. Li and J. Wang, "Fedmd: Heterogenous federated learning via model distillation," *arXiv preprint arXiv:1910.03581*, 2019.

[34] C. Wu, F. Wu, L. Lyu, Y. Huang, and X. Xie, "Communication-efficient federated learning via knowledge distillation," *Nature communications*, vol. 13, no. 1, p. 2032, 2022.

[35] T. Shen, J. Zhang, X. Jia, F. Zhang, G. Huang, P. Zhou, K. Kuang, F. Wu, and C. Wu, "Federated mutual learning," *arXiv preprint arXiv:2006.16765*, 2020.

[36] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton, "Large scale distributed neural network training through online distillation," *arXiv preprint arXiv:1804.03235*, 2018.

[37] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data," *arXiv preprint arXiv:1811.11479*, 2018.

[38] C. He, M. Annavaram, and S. Avestimehr, "Group knowledge transfer: Federated learning of large cnns at the edge," *Advances in neural information processing systems*, vol. 33, pp. 14068–14080, 2020.

[39] T. Lin, L. Kong, S. U. Stich, and M. Jaggi, "Ensemble distillation for robust model fusion in federated learning," *Advances in neural information processing systems*, vol. 33, pp. 2351–2363, 2020.

[40] Y. J. Cho, A. Manoel, G. Joshi, R. Sim, and D. Dimitriadis, "Heterogeneous ensemble knowledge transfer for training large models in federated learning," *arXiv preprint arXiv:2204.12703*, 2022.

[41] F. Sattler, T. Korjakow, R. Rischke, and W. Samek, "Fedaux: Leveraging unlabeled auxiliary data in federated learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 9, pp. 5531–5543, 2021.

[42] S. Cheng, J. Wu, Y. Xiao, and Y. Liu, "Fedgems: Federated learning of larger server models via selective knowledge fusion," *arXiv preprint arXiv:2110.11027*, 2021.

[43] L. Hu, H. Yan, L. Li, Z. Pan, X. Liu, and Z. Zhang, "Mhat: An efficient model-heterogenous aggregation training scheme for federated learning," *Information Sciences*, vol. 560, pp. 493–503, 2021.

[44] J. Zhang, S. Guo, X. Ma, H. Wang, W. Xu, and F. Wu, "Parameterized knowledge transfer for personalized federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 10092–10104, 2021.

[45] Y. J. Cho, J. Wang, T. Chirvolu, and G. Joshi, "Communication-efficient and model-heterogeneous personalized federated learning via clustered knowledge transfer," *IEEE Journal of Selected Topics in Signal Processing*, vol. 17, no. 1, pp. 234–247, 2023.

[46] C. Briggs, Z. Fan, and P. Andras, "Federated learning with hierarchical clustering of local updates to improve training on non-iid data," in *2020 international joint conference on neural networks (IJCNN)*, pp. 1–9, IEEE, 2020.

[47] A. Ghosh, J. Chung, D. Yin, and K. Ramchandran, "An efficient framework for clustered federated learning," *Advances in neural information processing systems*, vol. 33, pp. 19586–19597, 2020.

[48] F. Sattler, K.-R. Müller, and W. Samek, "Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 8, pp. 3710–3722, 2020.

[49] K. Singhal, H. Sidahmed, Z. Garrett, S. Wu, J. Rush, and S. Prakash, "Federated reconstruction: Partially local federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 11220–11232, 2021.

[50] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018.

[51] M. G. Poirot, P. Vepakomma, K. Chang, J. Kalpathy-Cramer, R. Gupta, and R. Raskar, "Split learning for collaborative deep learning in healthcare," *arXiv preprint arXiv:1912.12115*, 2019.

[52] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," *arXiv preprint arXiv:1812.00564*, 2018.

[53] K. Palanisamy, V. Khimani, M. H. Moti, and D. Chatzopoulos, "Spliteasy: A practical approach for training ml models on mobile devices," in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pp. 37–43, 2021.

[54] Q. Duan, S. Hu, R. Deng, and Z. Lu, "Combined federated and split learning in edge computing for ubiquitous intelligence in internet of things: State-of-the-art and future directions," *Sensors*, vol. 22, no. 16, p. 5983, 2022.

[55] V. Turina, Z. Zhang, F. Esposito, and I. Matta, "Combining split and federated architectures for efficiency and privacy in deep learning," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pp. 562–563, 2020.

[56] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "Splitfed: When federated learning meets split learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, pp. 8485–8493, 2022.

[57] Y. Gao, M. Kim, C. Thapa, A. Abuadbba, Z. Zhang, S. Camtepe, H. Kim, and S. Nepal, "Evaluation and optimization of distributed machine learning techniques for internet of things," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2538–2552, 2021.

[58] M. Gawali, C. Arvind, S. Suryavanshi, H. Madaan, A. Gaikwad, K. Bhanu Prakash, V. Kulkarni, and A. Pant, "Comparison of privacy-preserving

distributed deep learning methods in healthcare," in *Annual conference on medical image understanding and analysis*, pp. 457–471, Springer, 2021.

[59] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[60] A. Singh, P. Vepakomma, O. Gupta, and R. Raskar, "Detailed comparison of communication efficiency of split learning and federated learning," *arXiv preprint arXiv:1909.09145*, 2019.

[61] Y. Liao, Y. Xu, H. Xu, Z. Yao, L. Wang, and C. Qiao, "Accelerating federated learning with data and model parallelism in edge computing," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 904–918, 2023.

[62] A. Chopra, S. K. Sahu, A. Singh, A. Java, P. Vepakomma, V. Sharma, and R. Raskar, "Adasplit: Adaptive trade-offs for resource-constrained distributed deep learning," *arXiv preprint arXiv:2112.01637*, 2021.

[63] D. Pasquini, G. Ateniese, and M. Bernaschi, "Unleashing the tiger: Inference attacks on split learning," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pp. 2113–2129, 2021.

[64] E. Diao, J. Ding, and V. Tarokh, "Heterofl: Computation and communication efficient federated learning for heterogeneous clients," *arXiv preprint arXiv:2010.01264*, 2020.

[65] S. Horvath, S. Laskaridis, M. Almeida, I. Leontiadis, S. Venieris, and N. Lane, "Fjord: Fair and accurate federated learning under heterogeneous targets with

ordered dropout," *Advances in Neural Information Processing Systems*, vol. 34, pp. 12876–12889, 2021.

[66] S. Caldas, J. Konečny, H. B. McMahan, and A. Talwalkar, "Expanding the reach of federated learning by reducing client resource requirements," *arXiv preprint arXiv:1812.07210*, 2018.

[67] D. Wen, K.-J. Jeon, and K. Huang, "Federated dropout—a simple approach for enabling federated learning on resource constrained devices," *IEEE wireless communications letters*, vol. 11, no. 5, pp. 923–927, 2022.

[68] G. Cheng, Z. Charles, Z. Garrett, and K. Rush, "Does federated dropout actually work?," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 3387–3395, 2022.

[69] S. Alam, L. Liu, M. Yan, and M. Zhang, "Fedrolex: Model-heterogeneous federated learning with rolling sub-model extraction," *Advances in neural information processing systems*, vol. 35, pp. 29677–29690, 2022.

[70] Y. Niu, S. Prakash, S. Kundu, S. Lee, and S. Avestimehr, "Federated learning of large models at the edge via principal sub-model training," *arXiv preprint arXiv:2208.13141*, 2022.

[71] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2017.

[72] C. R. Wolfe, J. Yang, F. Liao, A. Chowdhury, C. Dun, A. Bayer, S. Segarra, and A. Kyrillidis, "Gist: Distributed training for large-scale graph convolutional networks," *Journal of Applied and Computational Topology*, vol. 8, no. 5, pp. 1363–1415, 2024.

[73] C. Dun, M. Hipolito, C. Jermaine, D. Dimitriadis, and A. Kyrillidis, "Efficient and light-weight federated learning via asynchronous distributed dropout," in *International Conference on Artificial Intelligence and Statistics*, pp. 6630–6660, PMLR, 2023.

[74] W. Fang, D.-J. Han, and C. G. Brinton, "Submodel partitioning in hierarchical federated learning: Algorithm design and convergence analysis," in *ICC 2024-IEEE International Conference on Communications*, pp. 268–273, IEEE, 2024.

[75] P. Goyal, "Accurate, large minibatch sg d: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[76] Y. You, I. Gitman, and B. Ginsburg, "Scaling sgd batch size to 32k for imagenet training," *arXiv preprint arXiv:1708.03888*, vol. 6, no. 12, p. 6, 2017.

[77] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pp. 583–598, 2014.

[78] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[79] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[80] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," *Advances in neural information processing systems*, vol. 31, 2018.

[81] S. Shi, X. Chu, and B. Li, "Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 172–180, IEEE, 2019.

[82] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze, "Plink: Efficient cloud-based training with topology-aware dynamic hierarchical aggregation," in *Proceedings of the 3rd MLSys Conference*, 2020.

[83] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2238–2247, IEEE, 2019.

[84] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi, "Error feedback fixes signsgd and other gradient compression schemes," in *International Conference on Machine Learning*, pp. 3252–3261, PMLR, 2019.

[85] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[86] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks.," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[87] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

[88] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using

pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[89] S. Tavarageri, S. Sridharan, and B. Kaul, "Automatic model parallelism for deep neural networks with compiler and hardware support," *arXiv preprint arXiv:1906.08168*, 2019.

[90] W. Zhu, C. Zhao, W. Li, H. Roth, Z. Xu, and D. Xu, "Lamp: Large deep nets with automated model parallelism for image segmentation," in *Medical Image Computing and Computer Assisted Intervention–MICCAI 2020: 23rd International Conference, Lima, Peru, October 4–8, 2020, Proceedings, Part IV 23*, pp. 374–384, Springer, 2020.

[91] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta, "Optimizing multi-gpu parallelization strategies for deep learning training," *Ieee Micro*, vol. 39, no. 5, pp. 91–101, 2019.

[92] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "{HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 307–321, 2020.

[93] X. Jia, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[94] S. Tyagi and P. Sharma, "Taming resource heterogeneity in distributed ml training with dynamic batching," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 188–194, IEEE, 2020.

[95] F. Chen, P. Li, C. Wu, and S. Guo, "Non-clairvoyant scheduling of distributed machine learning with inter-job and intra-job parallelism on heterogeneous gpus," *IEEE Transactions on Cloud Computing*, vol. 12, no. 4, pp. 1011–1025, 2024.

[96] R. Zhou, J. Pang, Q. Zhang, C. Wu, L. Jiao, Y. Zhong, and Z. Li, "Online scheduling algorithm for heterogeneous distributed machine learning jobs," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1514–1529, 2022.

[97] H. Sun, Z. Gui, S. Guo, Q. Qi, J. Wang, and J. Liao, "Gssp: Eliminating stragglers through grouping synchronous for distributed deep learning in heterogeneous cluster," *IEEE transactions on cloud computing*, vol. 10, no. 4, pp. 2637–2648, 2021.

[98] K. Osawa, S. Li, and T. Hoefler, "Pipefisher: Efficient training of large language models using pipelining and fisher information matrices," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 708–727, 2023.

[99] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 463–479, 2020.

[100] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[101] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[102] NVIDIA Corporation, *NVIDIA V100 Tensor Core GPU*, 2017. Accessed: 2024-12-09.

[103] Intel Corporation, *Intel Xeon Processor Scalable Family (Skylake)*, 2017. Accessed: 2024-12-09.

[104] NVIDIA Corporation, *NVIDIA T4 Tensor Core GPU*, 2018. Accessed: 2024-12-09.

[105] NVIDIA Corporation, *NVIDIA A2 Tensor Core GPU*, 2021. Accessed: 2024-12-09.

[106] NVIDIA Corporation, *NVIDIA Tesla P4 GPU*, 2016. Accessed: 2024-12-09.

[107] Intel Corporation, *Intel Core Processor Family (Broadwell)*, 2014. Accessed: 2024-12-09.

[108] Digital Research Alliance of Canada, "Arbutus cloud." `https://docs.alliancecan.ca/wiki/Arbutus`, 2025. Accessed: July 7, 2025.

[109] NVIDIA Corporation, "Nvidia tesla v100 gpu datasheet." `https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf`, 2017. Accessed: 2025-08-01.

[110] NVIDIA Corporation, "Nvidia t4 tensor core gpu datasheet." `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet.pdf`, 2018. Accessed: 2025-08-01.

[111] NVIDIA Corporation, "Nvidia a2 tensor core gpu datasheet." `https://www.nvidia.com/content/dam/en-zz/solutions/data-center/a2/pdf/a2-datasheet.pdf`, 2021. Accessed: 2025-08-01.

[112] NVIDIA Corporation, "Nvidia tesla p4 datasheet." `https://images.nvidia.com/content/pdf/tesla/`

184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf, 2016. Accessed: 2025-08-01.

[113] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018.

[114] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[115] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035, Curran Associates, Inc., 2019.

[116] P. Team, "Pytorch distributed training," 2024. Accessed: 2024-12-09.

[117] Q. Zhang, D. Ram, C. Hawkins, S. Zha, and T. Zhao, "Efficient long-range transformers: You need to attend more, but not necessarily at every layer," 2023.

[118] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems*, vol. 35, pp. 16344–16359, 2022.