David Weissteiner

Decentralized Federated Learning: Framework Design, Communication Efficiency, and Dynamic Synchronization

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Andreas Trügler, Priv.-Doz. Mag. Dr.rer.nat

Know Center Research GmbH and University of Graz

Graz, 12 2025

Abstract

Federated Learning enables collaborative machine learning in distributed data environments without consolidation of raw data. In Centralized Federated Learning, devices share updates to the model parameters with a central server that aggregates them and redistributes the aggregated state. However, the central server poses a single point of failure and a major communication bottleneck. Decentralized Federated Learning (DFL) overcomes these problems by eliminating the central role and distributing the responsibilities to individual devices (actors). In this decentralized setting, communication is no longer centered on one point, allowing actors to communicate arbitrarily with their peers. The amplified possibility of network links and potential device limitations, however, raises the challenge of communication efficiency. In particular in resource-constrained environments, heavy communication workloads slow down the entire system, sometimes even rendering the application infeasible due to bandwidth limitations. In this thesis, we approach the challenge of communication efficiency in DFL. We start by outlining the fundamentals of DFL to clarify the general paradigm and identify the key challenges. Finding that a DFL system consists of numerous components, we recognize the difficulty of comparing different methods of the individual components with each other. Therefore, we implemented MoDeFL, a **Mo**dular **De**centralized **F**ederated **L**earning framework to facilitate the evaluation of DFL methods and to foster their comparability. MoDeFL aims to enable complete configuration and independent interchangeability of individual component methods. The implementation of a multitude of communication optimization methods and the support of communication-relevant evaluation metrics in MoDeFL has sparked our interest in methods for communication efficiency. In response, we survey the DFL literature with a particular focus on communication efficiency. Beyond establishing a taxonomy for the various communication-efficient techniques in DFL, this survey reveals the topic of dynamic synchronization among the underexplored research directions in DFL. In light of this research gap, we design Gradient Thresholding, a novel synchronization algorithm to reduce communication cost while preventing actors from diverging. This algorithm is able to detect divergent actors without the need for additional communication and triggers the synchronization process accordingly. Thus, Gradient Thresholding reduces communication cost by omitting exchanges of model updates between actors. Our experiments demonstrate the superiority of Gradient Thresholding over baselines and further illustrate its concept.

Acknowledgments

Academic Guidance I would like to express my deepest appreciation to all those who have supported me throughout the process of preparing this thesis. I am particularly grateful to the following people, without whom I could not have undertaken this journey:

- Andreas Trügler for supervising this thesis and guiding my academic path.
- Lea Demelius for her support with the survey paper and for repeatedly reminding me of the scientific value.
- Roman Kern for providing immediate clarifications on spontaneous inquiries regarding scientific and academic procedures.
- Hussain Hussain for advices and discussions on scientific practices and graphs.
- Andreas Ofner for his support in improving and refining the visualizations.

Funding This work was supported by the "PRO'k'RESS" project (grant No. 60155996) and partly by the "QUICHE" project (grant No. 54741739), both funded by the Austrian Research Promotion Agency (FFG). A visit to the ELSA Workshop on Privacy-Preserving Machine Learning was reimbursed by the project European Lighthouse on Safe and Secure AI (ELSA; funded by the European Union; project grant No. 101070617) through the ELSA PhD & Postdoc Mobility Programme.

Employment The work on this thesis was conducted as part of my employment at Know Center Research GmbH. Know Center Research GmbH is a COMET competence center that is financed by the Austrian Federal Ministry of Innovation, Mobility, and Infrastructure (BMIMI), the Austrian Federal Ministry of Economy, Energy, and Tourism (BMWET), the Province of Styria, the Steirische Wirtschaftsförderungsgesellschaft m.b.H. (SFG), the Vienna business agency, and the Standortagentur Tirol. The COMET programme is managed by the Austrian Research Promotion Agency FFG.

Artificial Intelligence (AI) Support We acknowledge the use of the following AI tools in the writing process of this thesis for support on spelling correction and stylistic revision: DeepL Translate (`https://www.deepl.com/en/translator`), Grammarly (`https://www.grammarly.com/`, Perplexity (`https://www.perplexity.ai/`), and Writefull for Overleaf (`https://www.writefull.com/writefull-for-overleaf`).

AFFIDAVIT

Contents

1. Introduction

The Machine Learning paradigm Federated Learning (FL) has rapidly gained popularity since its inception in 2016 [1]. The properties of data locality and data privacy render FL particularly suitable for numerous distributed optimization applications. However, traditional FL relies on a parameter server architecture and, thus, on a central entity. Since this central server comprises a single point of failure and a great communication bottleneck, the FL paradigm of Decentralized Federated Learning (DFL) has been introduced. In the present thesis, we approach the challenge of communication efficiency in DFL, starting with the general DFL paradigm towards a novel method for reducing the communication cost in DFL. The following paragraphs outline the main chapters of this thesis.

Decentralized Federated Learning improves the robustness and scalability of traditional FL paradigms such as Centralized Federated Learning (CFL). Therefore, DFL has gained increasing popularity both in research and in industry. Although DFL overcomes certain limitations encountered in CFL, there still exist major challenges. The change in setting compared to CFL demands that these challenges be approached from a different perspective. In Chapter 2, we define the DFL paradigm and outline the main components of the system. We outline the key challenges in DFL research and present approaches from the literature that address these challenges. We then provide an overview of common application scenarios and their respective prevailing challenges. This chapter offers a high-level overview of DFL and related research, serving as an introduction to the research field of DFL.

To address the numerous challenges and trade-offs in DFL, applications typically combine various techniques of individual components. These components include, for example, the method for aggregating model parameters, the network topology, the local optimization step at the actor, etc. Evaluations of newly proposed methods for these components are usually conducted on a dedicated simulation platform and compared with selected baselines. However, the experimental results obtained in a custom framework impede the direct comparability of findings among different methods. We implemented a **Mo**dular **De**centralized **F**ederated **L**earning framework (MoDeFL) to facilitate experiments on different methods in a common framework. MoDeFL separates components into distinct modules that can be configured independently of each other in terms of the method applied and its hyper-parameters. Furthermore, it allows simple integration of novel techniques within these modules. Thus, MoDeFL provides a modular and customizable framework for the experimental comparison of different DFL techniques in a common environment. We elaborate on the system design of MoDeFL in Chapter 3.

DFL tries to eliminate the communication bottleneck and the single point of failure found in CFL. However, the challenge of communication efficiency in DFL persists due to the amplified number of possible network links, and due to systems constrained in their network resources. Despite efforts already invested in improving communication efficiency, the existing literature lacks a comprehensive overview of communication aspects in DFL. To fill this gap, we survey recent publications with a particular focus on communication efficiency in Chapter 4, explicitly addressing the communication cost of key DFL constituents. We establish a taxonomy of essential DFL components that influence communication by identifying classes of strategies and linking them to implementations in the current literature. Next, we examine popular techniques to optimize the communication cost in DFL and categorize these methods into sub-classes. Taking these components and optimization techniques into account, we then review recent publications in terms of communication efficiency and summarize their experimental results to assess the communication overhead. This survey provides an examination and categorization of key communication aspects in DFL, which is valuable for both researchers and practitioners engaged in resource-constrained distributed optimization.

Mutual exchange and incorporation of model parameter updates by actors is crucial to enable collaborative training and ensure that actors optimize for the common global objective in DFL. However, frequent synchronization of model updates increases the communication cost, as model updates are exchanged over the network. Therefore, improved synchronization algorithms are needed to optimize the trade-off between convergence and communication efficiency. In Chapter 5, we design Gradient Thresholding, a novel synchronization rule to reduce communication cost in DFL while preventing different actors from diverging. Based on the observation that actors only require synchronization if their model parameters diverge from each other, we construct a threshold region in the parameter space, defining the area in which actors can update their model parameters without the need for synchronization. Exceeding this threshold region indicates that the respective actor might diverge, triggering the synchronization of all actors. In the experiments, we demonstrate the superiority of Gradient Thresholding over selected baselines and support the concept with empirical illustrations.

Contributions Our contributions to the scientific community include the following:

- We implemented MoDeFL, a novel DFL framework to promote scientific experimentation with individual DFL methods. In this work, we share the system design of MoDeFL, providing valuable considerations for the modular implementation of DFL components and their consolidation into a complete system. Furthermore, we share the implementation of MoDeFL publicly on GitHub[1].
- We synthesize and analyze the current state of DFL research with regard to communication efficiency. We establish a taxonomy of DFL methods, unveil the impact of different methodological classes on communication, and provide detailed insights on actual implementations in recent publications, as well as discussions about the interaction of communication efficiency with other research challenges in DFL. In addition to being included in the present thesis, this item has been submitted for publication as a survey paper and is currently under review [2].
- We propose Gradient Thresholding, a novel synchronization rule to optimize the trade-off between communication efficiency and performance. We explain the methodology of Gradient Thresholding and provide a detailed description of the algorithm. We present the results of extensive experiments in which we confirm the superiority of Gradient Thresholding over baselines, demonstrate the concept of Gradient Thresholding in an experimental environment, and investigate its sen-

[1]MoDeFL GitHub repository: `https://github.com/ywcb00/MoDeFL/tree/v0.1.1`

sitivity to certain attributes and hyperparameters. The implementation of the algorithm and our experiments is publicly available on GitHub[2].

Structure The chapters of this thesis are structured in the following way: Chapter 2 elaborates on the fundamentals of DFL; Chapter 3 provides the system design of our implemented DFL framework MoDeFL; Chapter 4 comprises the survey on communication efficiency in DFL; and Chapter 5 introduces our novel synchronization rule Gradient Thresholding.

Publication and Author Contribution Chapter 4 and parts of Chapter 2 have been submitted for publication to the journal 'Artificial Intelligence Review' and are currently undergoing peer review [2]. David Weissteiner is the main author of the manuscript. Lea Demelius co-authored this work by contributing to the conceptualization, writing of privacy-related content, reviewing, and editing. Andreas Trügler co-authored this work by contributing to the conceptualization, reviewing, editing, and supervision.

[2]Gradient Thresholding GitHub repository: `https://github.com/ywcb00/GradientThresholding/tree/v0.1.1`

2. Decentralized Federated Learning: Fundamentals

The expanding field of machine learning (ML), with its constant desire for access to more data, is drawing increasing attention to data privacy considerations. The privacy constraints and regulations that arise from these considerations often prevent data transfer from edge devices or enterprise locations to a central location where traditional ML is performed. Therefore, Federated Learning (FL) has been proposed to overcome these restrictions without violating privacy constraints [1].

FL quickly acquired a wide range of applications, becoming relevant in various disciplines. However, the traditional paradigm of Centralized Federated Learning (CFL) faces limitations and weaknesses in certain applications due to its dependence on a central entity. Therefore, the paradigm of Decentralized Federated Learning (DFL) emerged to address these shortcomings. In this chapter, we elaborate on the fundamentals of Decentralized Federated Learning, including the general paradigm, the research challenges, and popular application scenarios.

2.1. Background

2.1.1. Centralized Federated Learning

CFL—as the primary variant of FL—enables model training at a centralized node (server) on decentralized data from multiple client devices (federated workers) with privacy constraints. This is achieved by utilizing the computing resources of the respective federated workers to compute pre-aggregates from the data, which do not allow for the recreation of the source data. Subsequently, the federated workers send their individual pre-aggregates to the centralized node (server), where they are aggregated and finalized to a global model, which is then returned to the federated workers (see Figure 2.1a). Thus, CFL addresses concerns about privacy, ownership, and locality of data [3].

Research in the field of CFL focuses mainly on four central challenges [4], [5]:

1. *Security and Privacy*: Although FL addresses some privacy concerns, research has shown that model updates can still reveal sensitive information about the underlying training data [6]–[8]. Therefore, CFL is often combined with other privacy-enhancing technologies, for example, differential privacy [9] or homomorphic encryption [10], [11]. In addition, security challenges arise, such as availability and integrity [12]. The central server represents a single point of failure, making it an attractive target for attacks.
2. *Data Heterogeneity*: Data partitions violate the assumptions about independent and identically distributed (I.I.D.) data in distributed optimization. For that reason, traditional CFL methods face the challenge of poor convergence on hetero-

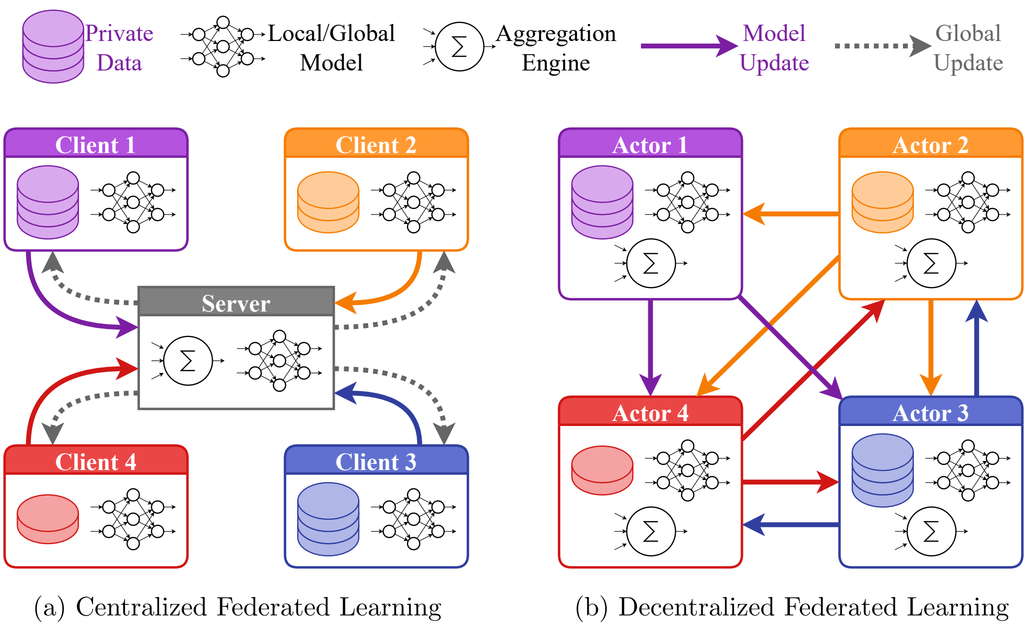


Figure 2.1.: Illustration of system architectures; in Centralized Federated Learning (Figure 2.1a), clients train their local model on private data and send model updates to the server which aggregates them into one global model update and distributes it to the clients; in Decentralized Federated Learning (Figure 2.1b), actors train their model on private data, transmit model updates to neighboring actors (arbitrary connections here), and aggregate received model updates to their local model.

geneous data. The majority of the existing work on preventing such convergence issues is based on techniques from data augmentation, client selection, regularization, meta-learning, and transfer learning [13].

3. *System Heterogeneity*: Federated workers often differ in their computing, communication, and storage resources as well as in their general availability. To mitigate the problems that arise from heterogeneous client resources, asynchronous CFL and semi-asynchronous CFL have been proposed [14].
4. *Communication Bottleneck*: All federated workers send their model updates to the same central server. Current approaches to reducing communication cost focus mainly on partial client participation, local updating, and compression techniques [15]–[17]. Furthermore, the paradigm of decentralized FL—on which we focus in the present work—aims to overcome the central communication bottleneck.

2.2. Paradigm

DFL is a variant of Federated Learning that enables devices to train collaborative ML models without centralized coordination. In the literature, DFL is also referred to as peer-to-peer FL (P2P FL), gossip FL, or distributed FL. Contrary to CFL, there is no distinction between the server and the clients in DFL. During DFL model training, actors share the same responsibilities of exchanging updates to model parameters with their respective neighbors and incorporating information from the received updates into the training process (depicted in Figure 2.1b). That way, DFL eliminates the need for a central entity and, thus, removes the single point of failure and the major communication bottleneck arising from the server in CFL.

Although the high-level objective of DFL remains similar to CFL—namely, training ML models in distributed environments subject to privacy constraints—, eliminating the central coordination entity entails some key conceptual changes. Due to the absence of a single point of communication, DFL requires dedicated strategies to handle the communication of model parameter updates between neighboring actors. In addition, every actor has to aggregate the updates from its neighbors autonomously. However, the neighborhood does not necessarily include all other actors, but only a subset. Therefore, we have to consider model aggregation with a limited set of model parameter updates in DFL.

Besides the absence of a central coordination entity, the system prerequisites in DFL are similar to CFL: Multiple devices with computing resources and private data are connected over the network. Based on these prerequisites, we can set up a DFL system which is characterized by three core components:

1. *Learning units* to enable model training of actors using their local data.
2. *Inter-node communication* to exchange updates to model parameters between neighboring actors.
3. *Aggregation engine* to incorporate received updates to model parameters into the learning process.

Together, these components represent the three stages that comprise DFL: (1) Local updating, (2) model update exchange, and (3) model aggregation. During training, the sequence of these stages is executed iteratively, with one pass through all three stages referred to as a communication round. Within these three segments, it is possible to implement additional techniques to optimize system behavior and overcome existing challenges.

2.3. Challenges

Research in the field of DFL involves many of the same core challenges as CFL. However, due to the difference in setting, we need to address these challenges from a different perspective. The following paragraphs elaborate on four key challenges in DFL.

Communication Efficiency Although DFL overcomes the central communication bottleneck of CFL, its distributed setting poses additional challenges in communication. Due to the absence of a single point of communication, actors must exchange model updates with their peers. As an implication, actors typically communicate with multiple other actors concurrently. Hence, DFL experiences a high total amount of communication through the network. Additionally, actor devices are rarely dedicated servers with high bandwidth, and often exhibit varying bandwidths among each other due to their geo-spatial distribution. It is therefore likely to face bandwidth limitations. Furthermore, network communication is considerably time-consuming, which is particularly important for time-critical applications. These aspects emphasize the challenge of communication efficiency and justify active research conducted on communication-efficient methods for DFL. We elaborate on techniques to address this challenge in Chapter 4.

Security and Privacy Although DFL eliminates server-related threats (e.g., single point of failure), it inherits all other security and privacy challenges of CFL, and often amplifies the respective risks due to lack of trust and the higher complexity of peer-to-peer communication [18]. Key threats include: (1) node failures or dropouts, (2) malicious actors (including model poisoning attacks), and (3) information leakage (including inference attacks on private data). The likelihood of node failures and dropouts is increased due to diverse device capabilities and network conditions, requiring robust fault-tolerant solutions that avoid model version inconsistency. Enforcing trust and validating nodes is also more challenging without a central server, leading to the adoption of cryptographic methods and blockchain technologies to authenticate participants and ensure data integrity. Poisoning attacks are a major concern in FL, as malicious nodes can inject manipulated model updates into the system to degrade performance or introduce backdoors. Without a central aggregator, detecting such poisoned updates is more difficult since each node only observes a subset of updates (with exemption of fully-connected networks). Apart from robust aggregation rules and verification of nodes, anomaly detection is applied to filter out malicious updates (see e.g., [19]). To protect against information leakage, DFL can be combined with privacy-enhancing technologies such as multi-party computation (MPC), differential privacy (DP) and homomorphic encryption (HE), though doing so is generally more challenging than in CFL due to its decentralized peer-to-peer architecture [18]. Multi-party computation [20] is employed to enable participants to jointly compute aggregates without revealing their individual model updates (see for example [21]). Differential privacy [22], [23] is widely used to prevent inference of individual data points from model updates or from the trained model itself. In DFL, the most prominent approach is local DP, where each node adds carefully calibrated noise to its model updates before sharing them with peers. Although well-aligned with decentralized systems, local DP often leads to significant loss of accuracy. To mitigate this, researchers have proposed a novel DP notion called network DP that leverages the properties of DFL to amplify privacy guarantees [24], [25]. Homomorphic encryption enables participants to perform computations on encrypted model updates. While HE is a prominent approach in CFL, its deployment in DFL is more challenging due to the difficulty of coordinating key management, calling for adaptions of standard HE such as multi-key HE (see e.g., [26]).

Data Heterogeneity Similarly to CFL, DFL faces the challenge of heterogeneous data distributions between actors, leading to performance degradation and convergence issues. The approaches to tackle this challenge are divided into two differing objectives: (1) Global convergence or (2) personalized models. Approaches with the goal of global convergence continue to aggregate diverging models at a higher level to reach convergence of the globally aggregated model. To achieve this, Wang *et al.* [27] incorporate synthetic data generation to locally re-balance training datasets. In [28], on the other hand, the authors utilize the techniques of gradient push and neighbor selection to enhance global performance on heterogeneous data. Additionally, several new or modified aggregation schemes evolved to address this challenge, such as model fusion through a model knowledge transfer algorithm [29], self-knowledge distillation [30], a localized primal-dual method [31], and the incorporation of synthetic data from other actors [32]. Personalized models (also called customized models), in contrast, encourage actors to learn different (i.e., personalized) models while still benefiting from the common basis among the individual models. Some existing approaches tackle this objective by introducing a penalty in the objective function [33], by re-weighting model updates based on their similarity to the individual local models [34], or by employing personalized adaptive masks to customize sparse local models [35].

System Heterogeneity Due to the heterogeneity of devices in computing resources and connectivity, the participating actors differ in (1) computation time to perform their local update, (2) communication reliability due to interrupted connections, and (3) network latency due to differing bandwidths. Therefore, the DFL system encounters problems of stragglers (i.e., slow devices) that slow down the global learning process. Asynchronous training processes help to overcome this problem by continuing the training without waiting for the other actors—ignoring stragglers in the system and eliminating respective idle times (see Section 4.2.2) [36]–[38]. Besides asynchronous DFL, the challenge of system heterogeneity is also addressed by semi-synchronized strategies, for instance, by allowing actors to continue their local training until the slowest actor completes its local update [39]. However, with both the asynchronous and the semi-synchronous methods, the issue of stale (i.e., outdated) model updates arises, since actors learn at different rates. On top of asynchronous training, Cao *et al.* [36] propose a dynamic adjustment of local training steps with probability-based neighbor selection to further optimize the learning process with heterogeneous devices. Alternatively, in [37], the authors implement asynchronous training in combination with a model selection strategy that is based on reinforcement learning and dynamic weighting of neighbor models. Apart from asynchronous DFL, research also addresses system heterogeneity in synchronous systems. In [40], the authors explicitly address unreliable communication by enabling the aggregation of partially received models while dynamically optimizing the mixing weights according to the reliability of the individual network link. Similarly to approaches in the asynchronous case, Liao *et al.* [41] employ heterogeneity-aware methods to control the amount of local update steps and to select neighbors in synchronous DFL.

Note that all previous challenges entail the cross-cutting challenge of maintaining model consistency and accelerating convergence within this distributed setting.

2.4. Application Scenarios

DFL emerges as a useful tool for various applications, thanks to its great scalability and privacy-preserving characteristic. Among the most popular application scenarios are healthcare [42]–[45], Internet of Things (IoT) [46]–[51], satellite networks [52]–[56], and vehicles [57]–[64].

Healthcare While ML shows great potential for reducing human error in healthcare, traditional ML techniques prevent collaboration between institutions due to privacy concerns. FL emerges as a viable solution by enabling collaborative ML without sharing raw data. Nevertheless, FL still faces security and privacy threats. In CFL, the central server has to serve as a trusted third party, which is difficult to find when several institutions are involved. DFL does not include a central server, allowing healthcare institutions to collaborate on a common optimization problem without relying on a single entity. The geo-spatial distribution between healthcare institutions promotes optimizing communication efficiency to overcome the burden of long communication times. The sensitive nature of data in healthcare applications underlines the importance of addressing the privacy challenge [43]–[45] to provide privacy guarantees and thus ensure that data remain undisclosed to other parties. In addition to DFL among institutions, the growing field of the Internet of Medical Things in healthcare is drawing increasing attention toward DFL on mobile and wearable devices. Due to varying computing and communication resources of these devices, the need arises to consider the challenge of system heterogeneity—alongside privacy and communication efficiency.

Internet of Things (IoT) The increasing importance of IoT as well as Artificial Intelligence led to the field of Artificial Intelligence of Things (AIoT). In AIoT, various IoT devices empowered with Artificial Intelligence interact with each other to solve a common objective. FL enables AIoT in large-scale, data privacy preserving scenarios. As the communication resources of IoT devices can be limited, constant communication with the central server in CFL creates a major bottleneck. Therefore, DFL emerges as a promising paradigm for AIoT applications. Given the typically dense connectivity of IoT networks, the DFL system comprises a multitude of communication links. Since extensive communication over a vast number of links would slow down the entire network, communication still needs to be optimized—either by communicating only over a fraction of available links or by reducing the size of communicated information. Therefore, the challenge of communication efficiency arises in IoT applications along with the two aforementioned challenges. DFL applications in IoT are confronted with a wide variety of devices that show different system characteristics, accentuating the challenge of system heterogeneity [46], [50]. Since these devices also differ in their data acquisition, the challenge of data heterogeneity becomes evident [46], [49], [51].

Satellite Networks Low Earth Orbit satellites have received increasing attention recently, partly due to their significant role in research on the sixth generation mobile network (6G). Recent advances in satellite networks draw growing interest toward arising opportunities for analyzing satellite sensor data using ML techniques. Since, for example, transferring satellite imagery data to a ground server would likely exceed bandwidth limitations due to the high data volume, FL offers a promising solution to avoid large data transfers. Furthermore, to prevent dependence on communication with a central server, which often suffers from intermittent connectivity in such scenarios, DFL emerges as a promising paradigm. Nonetheless, DFL applications in satellite networks face the situation of highly sporadic, irregular connectivity [52] and energy restrictions [53] in general. Here, reducing communication is crucial to complete the exchange of model updates among actors within a given time frame, as well as to minimize the energy consumption of wireless communication. Due to differences in cameras and sensors, satellite DFL applications also face the challenge of data heterogeneity [53], [55], [56].

Vehicles Traditional ML in vehicular networks encounters problems of privacy, limited communication, and restricted energy consumption. DFL emerges as a promising avenue to enable collaborative training of ML models in this setting. While 'vehicles' is a broad term, recent literature includes applications in unmanned aerial vehicles [58], [60], [61], [63], road vehicles [57], [62], and autonomous underwater vehicles [59]. These applications face sporadic and irregular connectivity due to the consistent movement of vehicles—similar to applications in satellite networks. For that reason, optimizing communication efficiency is important to leverage times of stable connectivity most efficiently. Alongside the challenge of communication efficiency, DFL in vehicles faces the challenge of privacy and security, particularly in military applications involving unmanned aerial vehicles [58].

3. Modular Decentralized Federated Learning Framework: System Design

3.1. Introduction

The architecture of a DFL framework comprises numerous system components, each of which offers several strategies. These components can be categorized by their main responsibility in the DFL process, including data loading, network communication, model training, and aggregation of model parameters. The category of network communication, for instance, comprises the protocol used to transmit data over network, the network topology, and the listening service with the buffer for incoming messages, as well as potential optimization techniques such as compression and partial device participation.

Novel methods in DFL research are typically evaluated in a custom implementation tailored to their needs. However, the broad landscape of components and their strategies often leads to ambiguity in the experimental setup, as not all components are clearly specified. Furthermore, DFL approaches differ in the strategies of their components, hindering the comparison of the experimental results between different publications. These two aspects complicate the implementation of baseline comparisons with related approaches. This motivates us to implement MoDeFL, a modular DFL framework in which each module corresponds to a specific component. These modules implement individual strategies for the corresponding components, allowing us to specify the desired strategy under test without modifying other modules. This design thus facilitates the experimental comparison of different strategies for a specific component within the same system configuration.

In this chapter, we describe the high-level system design of MoDeFL. We start by defining the abstractions used throughout the chapter as well as in the implementation of the framework in Section 3.2. We proceed by describing the loading and partitioning of data (Section 3.3) as well as the construction of the ML model to perform local training (Section 3.4). Following this, we elaborate on the handling of the model parameters and gradients in MoDeFL in Section 3.5, including their representation, serialization, and aggregation. We then focus on network communication in Section 3.6, encompassing the network topology, the communication between actors, and the management of received network messages, as well as some techniques to optimize communication in DFL. After discussing all these constituents, we then provide a general view on the execution of the entire learning procedure in Section 3.7, and guide through available configurations in Section 3.8. The implementation of MoDeFL is publicly available on GitHub[1].

[1]MoDeFL GitHub repository: `https://github.com/ywcb00/MoDeFL/tree/v0.1.1`

3.2. Abstractions

The key constituents of a learning procedure in MoDeFL are the Initiator process responsible for system initialization, the actors that perform the actual training, and the model updates exchanged between actors to enable collaboration. Here, we outline these three abstractions to clarify what they refer to and what they entail.

3.2.1. Actor

The actor in MoDeFL represents the binding of private data, a local ML model, computation resources, and network resources. In a real-world deployment, an actor could be, for instance, a smartphone with location data (i.e., private data) that is connected to a wireless router. During the training phase (see Section 3.7.2), the actors are responsible for performing three essential tasks:

1. Local Training: The actors leverage their computing resources to train the local model on their respective private data (see Section 3.4.4).
2. Update Exchange: The actors exchange their updated model state (i.e., the model update; Section 3.2.3) over the network.
3. Aggregation: Actors aggregate the received model updates and update their local model accordingly (see Section 3.5.3).

By performing these three tasks repetitively, actors collaborate in training the ML model.

In MoDeFL, each actor is started as a stand-alone process which constantly listens for network messages from its respective neighboring actors. It is therefore possible to deploy actors on different machines, spatially separated from each other but connected over the network. Furthermore, MoDeFL allows for the deployment of an arbitrary number of actors. Apart from the Initiator (see Section 3.2.2), the actors are the only participants in our DFL system.

3.2.2. Initiator

Before starting the training procedure, the actors have to be initialized in order to collaborate on the global optimization problem. This initialization comprises informing an actor about his identity in the system (e.g., its public network address), setting up the network topology, setting the seed for reproducibility, specifying the dataset to be used, communicating which strategies are applied, and initializing the model in terms of model architecture, optimizer, and initial model weights. For this purpose, MoDeFL spawns a dedicated process (i.e., the Initiator) at system startup to carry out the initialization of the actors. The Initiator must show a network connection to all the actors, as it coordinates the initializations individually through network messages. More detailed information on the entailed network communication and the initialization phase is given in Section 3.6.2 and Section 3.7.1, respectively. Although the Initiator operates as a central node in the system, it does not contradict the decentralization of the DFL paradigm, since it only participates in the initialization phase. Thus, the Initiator terminates before the actual training phase begins, when all actors are initialized with the correct attributes.

3.2.3. Model Update

The model update (also called the model parameter update) refers to the information about the local enhancement of an actor's local model. It comprises the network message that is sent from one actor to the neighboring actors in each communication round (i.e., in each global epoch). Accordingly, each actor receives several model updates in each communication round, aggregates them, and incorporates the aggregation into the training. The representation of the model update depends on the overall DFL strategy; It can consist, for instance, of the complete set of model parameters, a gradient, or the difference of the local model parameters to some common reference (i.e., a model delta). In addition to this update of the local model, MoDeFL allows the use of aggregation weights in the model updates to weight the impact of individual actors when aggregating the received model updates. We further elaborate on the exchange of model updates in Section 3.6.2.

3.3. Data

MoDeFL focuses on supervised learning tasks and, therefore, supports data records in the format of (X, Y) tuples with X being the input features and Y being the output labels. Consequently, the term dataset refers to a collection of such data records. During the training phase (see 3.7.2), each actor trains the model on its own local dataset. This local dataset is either an independent dataset that has always resided with the corresponding actor (as in a real-world scenario), or a partition of a single, complete dataset that was partitioned and distributed to the actors (the simulated scenario; see 3.3.2). In the following subsections, we describe how a dataset can be loaded in MoDeFL and how it is partitioned to simulate the decentralization of data among actors.

3.3.1. Data Loading

MoDeFL provides the class interface `IDataset` to consolidate the heterogeneous landscape of data sources into a common representation. Implementing this interface ensures that datasets can be processed consistently within the framework. For this purpose, the `IDataset` interface specifies the class attributes `train`, `val`, and `test` to store the datasets for training, validation, and testing, respectively. The abstract method `load()` is responsible for loading the datasets and assigning them to the corresponding class attributes.

To add a new dataset to the DFL framework, we create a separate class that implements the `IDataset` interface, reflecting the desired dataset. Accordingly, we implement the corresponding `load()` method. Since we implement this method for each dataset individually, there are no restrictions on the way we load the data. The only requirement on the `load()` method is to supply the three datasets (i.e., train, validation, and test) in the format of TensorFlow [65] datasets. This means that we can either load the data as a TensorFlow dataset directly or in any other format, which is then converted into a TensorFlow dataset. Thus, MoDeFL enables various options for data loading, for instance, by using the TensorFlow Datasets (TFDS) collection [66] or by loading local files from disk.

3.3.2. Data Partitioning

Real-world DFL applications train an ML model on decentralized data that reside on multiple actors. Here, data on a specific actor is considered a data partition from the full dataset (i.e., all data partitions consolidated). Research experiments on DFL often simulate decentralized data by partitioning a single, large dataset into several partitions and distributing them to individual actors. This enables better control over the characteristics of the data partitions. Moreover, it allows for comparisons to traditional (i.e., centralized) ML methods and opens the possibility of comparing different levels of heterogeneity across the data partitions in the experiments.

In MoDeFL, the `FedDataset` class manages the partitioning of a given dataset into data partitions to simulate decentralized data. Once the complete dataset has been loaded as described in Section 3.3.1, we can instantiate the `FedDataset` class and pass the dataset to its `construct()` method. Subsequently, the dataset is partitioned into the same number of partitions as there are actors in the system, using the specified partitioning scheme[2]. MoDeFL supports the following partitioning schemes built-in:

- *Range:* Divide the dataset into the desired number of connected partitions, preserving the order of the data records (Figure 3.1a).
- *Random:* Assign each data record randomly to a partition (Figure 3.1b).
- *Round-Robin:* Assign data records to partitions sequentially in a cyclic manner (Figure 3.1c).
- *Dirichlet:* Allocate a proportion of data records of each label (applicable only for discrete responses) to each partition according to the Dirichlet distribution (see e.g., [68]; Figure 3.1d).

Further partitioning schemes can be easily added to MoDeFL by creating a new class method for partitioning within the `FedDataset` class. The requirements for such a partitioning method are to take a TensorFlow dataset and return the specified number of partitions as a list of TensorFlow Datasets.

[2]The partitioning scheme defines the way in which data records are divided into distinct partitions [67]

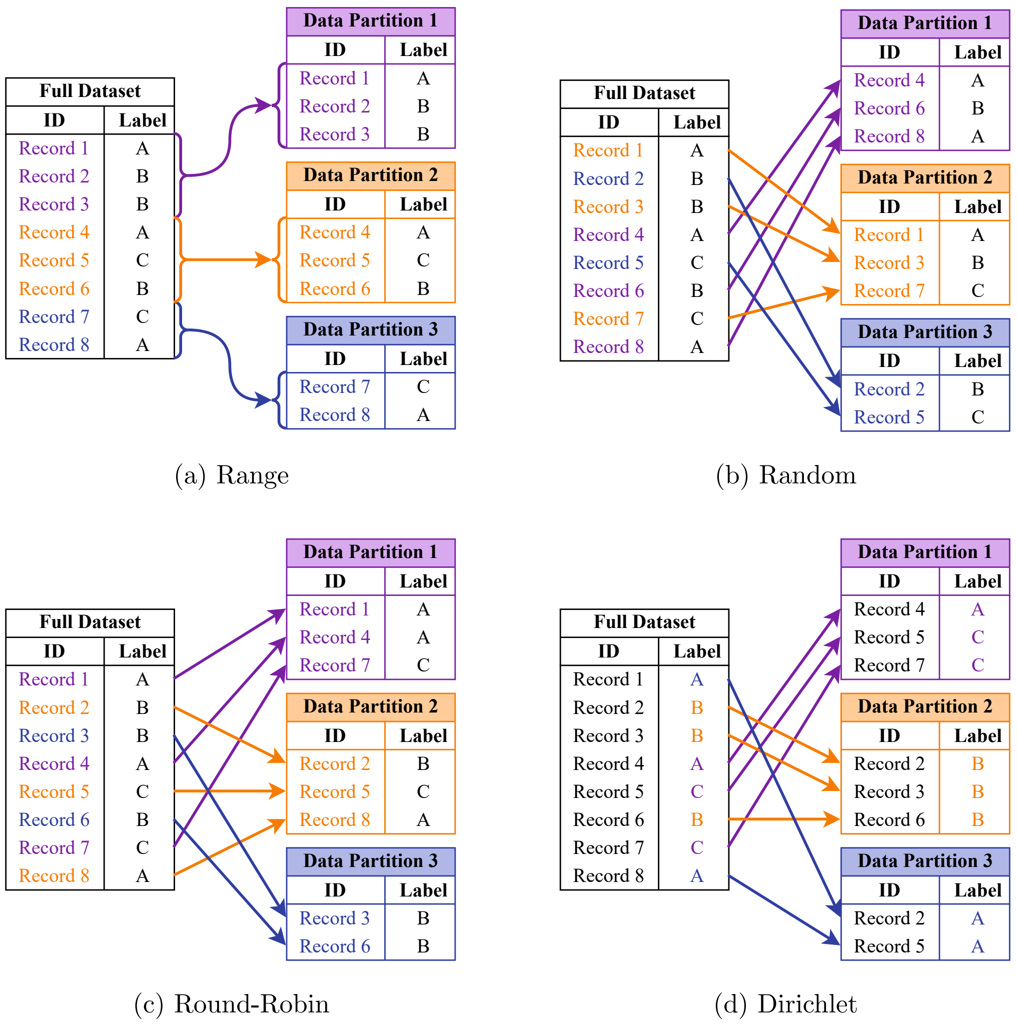


Figure 3.1.: Built-in partitioning schemes in MoDeFL: ‘Range’ partitioning (Figure 3.1a) preserves the order of the records from the original dataset across and within the partitions; ‘Random’ partitioning (Figure 3.1b) randomly assigns the records to equally sized partitions; ‘Round-Robin’ (Figure 3.1c) circulates the partitions to assign the records sequentially; ‘Dirichlet’ partitioning (Figure 3.1d) assigns the records based on their label to introduce a specific level of heterogeneity between partitions.

3.4. Model

The model constitutes the core part of the DFL system, whose objective is to solve the respective optimization task. It consists of a fixed model architecture with trainable parameters, which we intend to improve by training on the respective training data. In MoDeFL, the model is replicated on every actor. Thus, we do not consider the decentralized view when explaining the model in the following subsections, since only the model parameters are aggregated decentralized but the model itself is replicated. Instead, we describe the construction, handling, serialization, and training of the model from a local perspective. Note that the model parameters are discussed separately with a decentralized view in Section 3.5.

3.4.1. Model Builder Interface

The class interface `IModelBuilder` in MoDeFL serves as a template for specifying the model architecture. Every definition of a model architecture requires an individual model builder class which is an instance of `IModelBuilder`. This class contains the definition of the model architecture together with the method `buildModel()` to build the model and return the corresponding object. In addition, the model builder class specifies model-related properties such as the default learning rate, the loss function, and evaluation metrics.

To define a new model architecture, we create a model builder class that implements the `IModelBuilder` interface. In this class, we define the `buildModel()` method to instantiate the underlying model object. Furthermore, we implement the methods `getLoss()` and `getMetrics()` which return the functions to calculate the loss and evaluation metrics, respectively. We also set the learning rate as a class member in the constructor. Note that MoDeFL currently supports Keras [69] models for the underlying model object. Additional support for arbitrary types of models can be added by implementing the respective abstractions from the model interface discussed in Section 3.4.2.

3.4.2. Model Interface

A model class in MoDeFL serves as a container for the underlying model object, specifying generic methods for various types of underlying model. The class interface `IModel`, which must be implemented by all model classes, defines the following abstract methods:

- `fit(data)`: Perform a training step of the model on the given data.
- `predict(data)`: Return forecasts for given data.
- `evaluate(data)`: Evaluate the model using given data and return a dictionary of the metrics specified in the respective model builder (see Section 3.4.1).

Unlike the model builder, the model class does not depend on the model architecture but on the type of the underlying model. For underlying model objects of the Keras model type, for instance, the `KerasModel` class implements the three methods defined in `IModel`. Therefore, the model class consolidates the training functionality of an underlying model object to three general methods, creating a common representation among all types of models in MoDeFL.

Support for arbitrary types of underlying model objects can be added by creating a new model class that implements the `IModel` interface. Note that the model builder has to construct the correct type of underlying model object to match the model class. Implementing the three common methods for model training allows for direct integration into MoDeFL. To add support for PyTorch [70] models, for example, we create a new model class `PyTorchModel` that inherits from the `IModel` interface. Subsequently, we implement the three methods `fit(data)`, `predict(data)`, and `evaluate(data)`, which essentially translate the calls to the corresponding functionalities of the underlying PyTorch model. To eventually use the `PyTorchModel` class, we have to utilize a model builder that constructs an underlying PyTorch model object. This model object gets wrapped into an instance of `PyTorchModel`, which then provides the accessibility required for MoDeFL.

3.4.3. Model Serialization

Actors in DFL require common initialization in terms of model and network characteristics before starting the training procedure, as mentioned in Section 3.2.2. This initialization can be achieved either by starting all actors with the same initialization or by initializing the actors after startup from a central node. MoDeFL supports the initialization of the model architecture and the optimizer over network by the Initiator. To achieve this, both the configuration of model architecture and the configuration of the optimizer are serialized to their byte representation. Subsequently, the Initiator transmits these configurations to all actors in the DFL system during the initialization phase (see Section 3.7.1). The actors then initialize their model according to the received configurations, resulting in the same initial model architecture and optimizer on all actors.

3.4.4. Local Training

In MoDeFL, every actor holds a local ML model that is wrapped in an instance of the respective model class (see Section 3.4.2). During local training, actors enhance their local models by performing one or multiple rounds of training on their local data. This is done by calling the `fit(data)` method of the model class, which then conducts the training and updates the parameters of the local model. The exact fitting process is therefore defined by the implementation of the model class. In the built-in model class `KerasModel`, for instance, the `fit(data)` method executes the `fit()` method of the underlying Keras model object to train the model and update its parameter in one call. In addition to this basic method `fit(data)`, the `KerasModel` class also provides the method `fitGradient(data)` to train the model, update the model parameters, and return the gradient of the training process, as well as the method `computeGradient()` to solely obtain the next gradient without advancing the model parameters. Thereby, it creates the possibility to apply various local training strategies by calling or overriding the respective methods.

3.5. Model Parameters and Gradient

The model parameters and gradients differ among ML libraries in terms of their representation and usage. To support the use of arbitrary ML libraries in MoDeFL, we create custom data structures for model parameters and gradients with dedicated methods for modifying, combining, and serializing. These data structures inherit from the common base class `HeterogeneousArray` which defines basic abstract methods and fundamental functionality such as abstract serialization methods and essential linear algebra operations. In the following subsections, we elaborate on (1) the storing and representation of our custom parameters object, (2) the serialization of model parameters for network transmission, and (3) the modification and aggregation of model parameters.

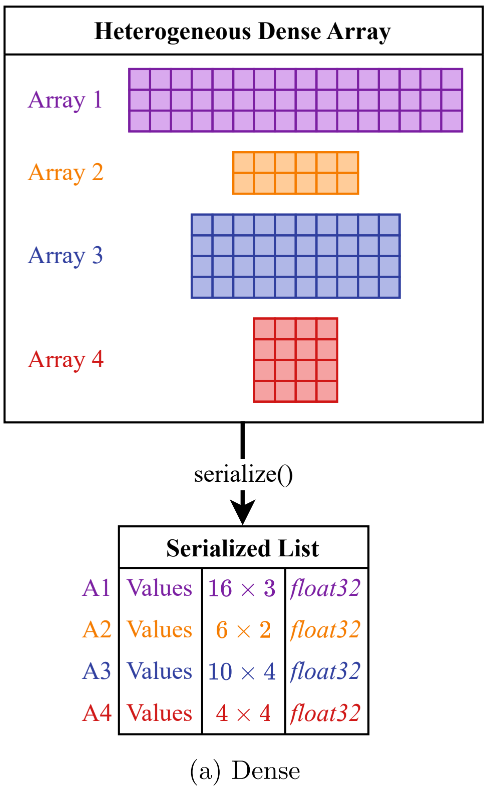
3.5.1. Representation

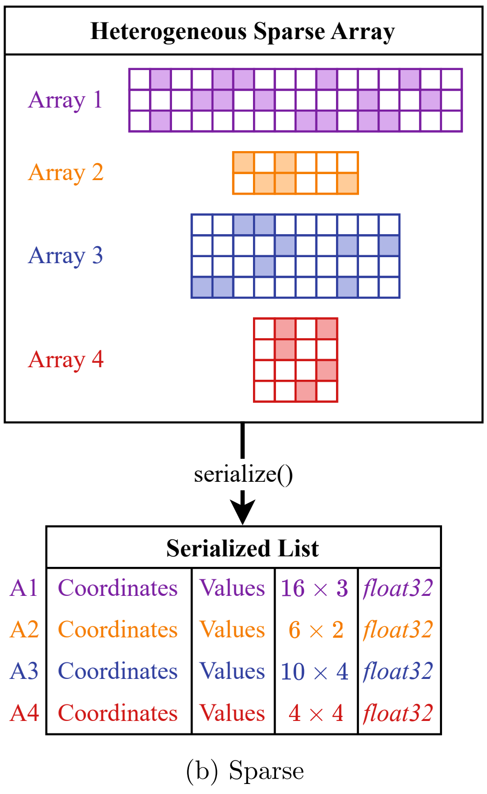
The `HeterogeneousArray` is an abstract class that serves as a common base class for all data structures used to store model parameters and gradients (in this section commonly referred to as parameters) in MoDeFL. We provide support for storing the parameters as both dense and sparse arrays, using the built-in classes `HeterogeneousDenseArray` and `HeterogeneousSparseArray`, respectively. The `HeterogeneousArray` stores parameters as a list of either Numpy [71] arrays in the dense case or sparse [72] arrays in the sparse case, along with some meta-data[3]. These arrays represent, for instance, the layer-wise parameters of a feed-forward neural network. On top of these underlying structures, the classes `HeterogeneousDenseArray` and `HeterogeneousSparseArray` implement methods for binary operations including addition, subtraction, multiplication, division, and comparison, as well as unary operations for taking the minimum, taking the maximum, and element-wise rounding down the values. Furthermore, they provide functionality to convert between dense and sparse representations. Through all these methods, the `HeterogeneousArray` classes enable simple yet complete operation for our application.

3.5.2. Serialization

In order to include the parameters in the model update (see Section 3.2.3) for exchange among the actors, they require a serialized format. The methods `serialize()` and `deserialize()` from the `HeterogeneousArray` classes perform the task of serializing the parameters to a byte representation and deserializing them back to an instance of `HeterogeneousArray`, respectively. To achieve this, we serialize the arrays stored inside the `HeterogeneousArray` object into a list of their serialized representations. One serialized representation is a consolidation of three entries for `HeterogeneousDenseArray` objects and four entries for `HeterogeneousSparseArray` objects, as illustrated in Figure 3.2. For both types, the last two entries of this consolidation contain the shape of the array, converted to bytes using the Numpy `tobytes()` method, as well as the data type of the array elements, encoded as a UTF-8 string. These two entries are required for deserializing the parameters back to an object of type `HeterogeneousArray`. For `HeterogeneousDenseArray` objects, the first entry in the serialized representation contains the values of the array. For `HeterogeneousSparseArray` objects, on the other hand, the first entry contains the coordinates of non-zero array elements and the second entry holds the corresponding element values. In both cases, these arrays are converted to a byte sequence using the Numpy method `tobytes()`. Thereby, we obtain a list of serialized representations of the parameters. In addition, we append potential meta-information about the parameters to this list, utilizing the library pickle [73] to convert this information into a byte object. This general format allows the transmission of parameters to other actors over the network. To convert the parameters back to the type of `HeterogeneousArray`, we iterate through the serialized list, converting each member back to its original array type with the correct shape and data type. As a result, we obtain the initial list of either Numpy arrays of sparse arrays, with which we instantiate the `HeterogeneousDenseArray` or the `HeterogeneousSparseArray`, respectively.

[3]In the case of compressed parameters (see Section 3.6.4), the `HeterogeneousArray` also contains meta-information about its compressed state (e.g., quantization bins).





Heterogeneous Sparse Array
Array 1
Array 2
Array 3
Array 4
serialize()
Serialized List
A1 Coordinates Values 16 × 3 float32
A2 Coordinates Values 6 × 2 float32
A3 Coordinates Values 10 × 4 float32
A4 Coordinates Values 4 × 4 float32
(b) Sparse

Figure 3.2.: Exemplary model parameters with serialized representations, which is a list containing the values, dimension, and datatype of each array from the source representation. For sparse arrays (Figure 3.2b), each list element additionally holds the coordinates of the non-zero values.

3.5.3. Aggregation

The aggregation of model updates from several actors is essential in DFL to incorporate the model enhancements from other actors into the learning process. Alongside the most popular averaging of model parameters, there exist numerous aggregation strategies. The static class `AggregationUtils` in MoDeFL the following built-in aggregation strategies:

- *Averaging*: Average of model parameters or gradients, optionally weighted by specified aggregation weights.
- *Consensus-based Federated Averaging*: Extension of the averaging strategy proposed by Savazzi *et al.* [74], which additionally weights received model updates by a common parameter to control the consensus rate.
- *Consensus-based Federated Averaging with Gradient Exchange*: Extension of Consensus-based Federated Averaging, which appends a second stage where actors also incorporate gradients computed from the aggregated model parameters on the data from their neighboring actors [74].
- *FedNova* Average of normalized stochastic gradients proposed by Wang *et al.* [75], in which the accumulated gradients are rescaled to eliminate inconsistencies in the solution.

These methods accept our custom types of model parameters and gradients (see Section 3.5.1), leveraging their built-in operational methods to compute aggregates. Support for further aggregation strategies can easily be added by creating a new method inside the `AggregationUtils` class.

3.6. Network Communication

The actors in DFL are individual devices that communicate over network links. Therefore, each actor contains a network interface card (NIC) through which it is connected to other actors. The set of actors that can be reached via an actor's NIC is determined by the network topology (see Section 3.6.1). To exchange information among each other, the actors must follow a common protocol for inter-node communication (see Section 3.6.2). Furthermore, actors require functionality to store and manage received network messages (see Section 3.6.3), since model updates are not processed instantly upon receipt.

3.6.1. Network Topology

The network topology determines which actors are connected to each other in our DFL system. In general, we allow for arbitrary, bidirectional network topologies in MoDeFL. Nevertheless, the specified topology must comply with the other parts of the system. For example, combining a ring topology, where each actor is connected to only two other actors, with a partial device participation strategy (see Section 3.6.4), which removes two network links from each actor, would isolate each actor and is therefore incompatible. Therefore, we must consider compatibility with the complete system when determining the network topology. Note that MoDeFL can be easily extended to support directed network topologies by storing the incoming and outgoing neighbors separately on each actor. In MoDeFL, we can specify the network topology in the form of an adjacency matrix, which is described in Section 3.8.1.

3.6.2. Inter-node Communication

The nodes in MoDeFL (i.e., the Initiator and the actors) communicate with each other using the remote procedure call framework gRPC [76]. In gRPC, the interfaces are defined using Protocol Buffers [77]—a data format to serialize structured data. Since we can divide the communication in our DFL system into an initialization phase (see Section 3.7.1) and a training phase (see Section 3.7.2), we also define our interfaces in two different services: the initialization service and the model update service. Although the procedures must be strictly predefined for gRPC, our aim is to keep their definitions as generic as possible to allow the implementation of a variety of DFL techniques. We elaborate on the procedures in the following paragraphs.

Initialization Service The initialization service provides remote procedures to specify the configuration and define the initial state of the actors. As explained in Section 3.2.2, MoDeFL spawns an initialization process (i.e., the Initiator) to initialize the individual actors before starting the training procedure. For that reason, each actor starts an initialization service that is permanently listening for calls to its predefined remote procedures. These initialization procedures comprise:

- `InitIdentity` tells the actor its public network address, its identifier among the actors, the number of actors in the system, and its individual seed to initialize the random generator.
- `InitDataset` specifies the dataset and the data partition to be loaded.
- `InitModel` transfers the serialized model to the actor and specifies the configuration for the optimizer.
- `InitModelParameters` initializes the model parameters at the actor.
- `InitStrategy` specifies the methodological strategies to be applied.
- `RegisterNeighbors` informs the actor about its respective neighbors.
- `StartLearning` terminates the initialization service and starts the training procedure.

The actor responds with an empty message as an acknowledgment of all calls to these procedures. Using these remote procedures, we can initialize the entire DFL system before starting the training procedure.

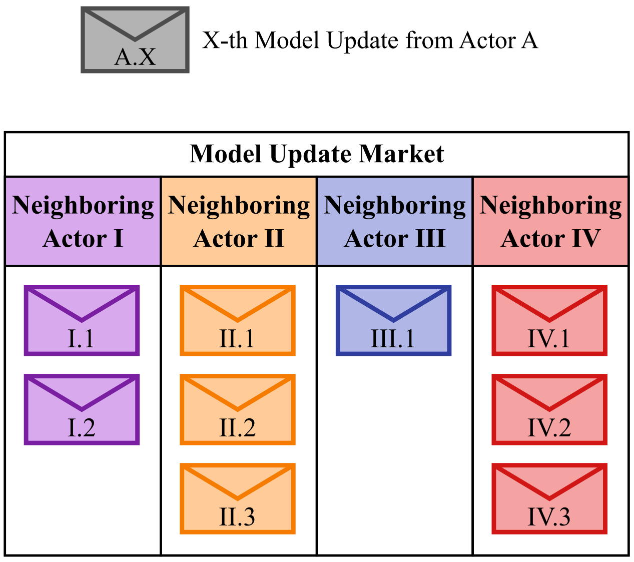
Model Update Service The model update service provides communication functionality during training. Once all actors have been initialized, the Initiator concludes the initialization phase by calling the `StartLearning` procedure on all actors. Consequently, each actor terminates the initialization service and starts the model update service. The model update service then permanently listens for calls to its predefined remote procedures, which include:

- `TransferModelUpdate` sends a model parameter update to the respective actor.
- `EvaluateModel` transfers the model parameters to the respective actor with the request to evaluate these parameters on its local data and to return the evaluation metrics.
- `AllowTermination` signals an actor that it will not receive any further calls from the calling node.

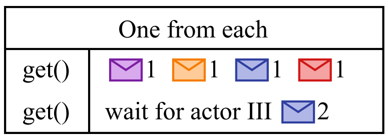
Note that the `TransferModelUpdate` procedure enables the exchange of model updates (e.g., model parameters, gradients) among actors and, hence, constitutes an essential piece of the DFL system. The `EvaluateModel` procedure enables the evaluation of model parameters of an actor on the private data of its neighboring actor. The `AllowTermination` procedure helps the actor terminate without prematurely closing the connection to its neighbors. Together, these remote procedures offer the communication ability to conduct the learning procedure in MoDeFL.

3.6.3. Model Update Market

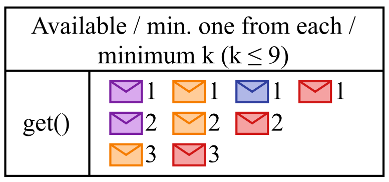
The model update market acts as a buffer for received model updates and enables various strategies for the consumption of model updates by the respective actor. Upon receiving a model update, the actor places it directly in its model update market through the `putUpdate()` method, where it is stored in a queue linked to the corresponding sender. Hence, the model update market comprises one queue for each neighboring actor, con-



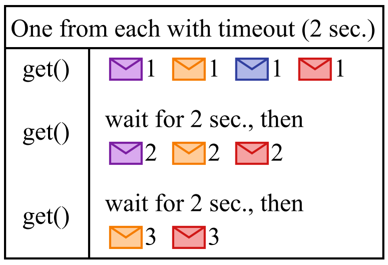
(a) Model Update Market



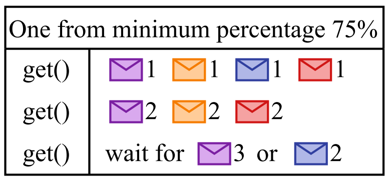
(b) One from each



(d) Available / Minimum one from each / Minimum k ($k \leq 9$)


One from each with timeout (2 sec.)
get() 1 1 1 1
get() wait for 2 sec., then 2 2 2
get() wait for 2 sec., then 3 3

(c) One from each with timeout



(e) One from minimum percentage

Figure 3.3.: Exemplary scenarios illustrating the strategies for retrieving model updates from the market (Figure 3.3a) via multiple invocations of the `get()` method.

taining the respective model updates received. Leveraging this data structure, the model update market provides the `get()` method to remove and return the model updates according to a specified strategy (i.e., the synchronization strategy). MoDeFL supports the following built-in strategies:

- *One from each*: Obtain one model update from each neighboring actor. Block if necessary until one model update is available for each neighboring actor.
- *Available*: Obtain all model updates available at the time.
- *Minimum one from each*: Obtain all model updates available at the time, but at

least one from each neighboring actor. Block if necessary until one model update is available for each neighboring actor.

- *One from minimum percentage*: Obtain one model update from at least a certain percentage of neighboring actors. Block if necessary until the percentage of model updates from neighboring actors is reached.
- *Minimum k*: Obtain all model updates available at the time, but at least k in total. Block if necessary until k model updates are available.
- *One from each with timeout*: Obtain one model update from each neighboring actor. Block if necessary until either one model update is available for each actor or the timeout is reached.

These strategies determine the synchronization behavior of the DFL system; the strategy *"one from each"*, for instance, refers to a strictly synchronized DFL system, whereas the strategy *"available"* enables completely asynchronous execution of the actors, as it does not include waiting times. Therefore, these strategies are also known as 'synchronization strategies'. To support additional strategies, we can extend the static class `ModelUpdateMarket` in MoDeFL by the respective method for obtaining the model updates.

3.6.4. Communication Optimization

Techniques for optimizing communication cost can be activated in addition to the required networking functionality that enable communication between the actors. Since these optimization techniques are optional, they can also be completely deactivated alongside the regular strategy options for modules in MoDeFL. Built-in communication optimization modules in MoDeFL comprise compression techniques and partial device participation strategies.

Compression Model updates in DFL are often compressed before being transferred over the network to reduce communication cost. More specifically, each actor applies lossy compression methods to the model parameters or gradient in a model update before distributing the model update to its neighboring actors. The most prominent compression methods in DFL are quantization and sparsification. Quantization methods reduce the number of bits required to represent the individual values of the model parameters or gradient. Sparsification methods, on the other hand, convert the model parameters or gradient into a sparse representation by introducing a large proportion of zero values, thereby reducing the number of values to transmit. MoDeFL supports the following built-in methods for quantization and sparsification:

- *Probabilistic Quantization*: Quantize the model parameters or gradient by probabilistically rounding down or up to a pre-specified precision of the values (similar to probabilistic quantization in Konecný *et al.* [78]).
- *Layer-wise TopK Sparsification*: Keep only the k highest values in each layer of the model parameters or gradient; remove (i.e,. zero-out) the other values.
- *Layer-wise Percentage Sparsification*: Keep only a specified percentage of highest values in each layer of the model parameters or gradient; remove the other values.

Partial Device Participation MoDeFL supports the communication optimization technique of partial device participation (PDP), which allows only a selected subset of neighboring actors to participate in each round. Consequently, the model updates are broadcast only to the selected neighbors, while the other neighbors receive an empty update message for synchronization purposes. MoDeFL currently provides one built-in PDP strategy: Randomly selecting a predefined number of neighbors for the transmission of model updates. However, the framework can be easily extended to support additional, more sophisticated strategies.

3.7. Execution Phases

The learning procedure in MoDeFL consists of three conceptual phases: Initialization, Training, and Evaluation. The initialization phase is managed by the Initiator (see Section 3.2.2), as defined in the `Initiator` class. The training and evaluation phases, on the other hand, are conducted by the actors and generally defined in a class that inherits from the `IDFLStrategy` interface class, referred to as 'learning strategy'. This allows the implementation of different learning strategies, which can be selected by the Initiator during the initialization.

3.7.1. Initialization Phase

The initialization phase revolves around initializing the attributes of the actors. First, all actor processes, as well as the Initiator, are started on their respective sites. Once all actors are running, the Initiator begins initializing the actors using the initialization service described in Section 3.6.2. This initialization informs the actors about their identity, their dataset, the model architecture and optimizer, the initial model parameters, the learning rate, the number of epochs, the general strategies to be used, and their neighboring actors in the network topology. The actors subsequently load the specified dataset and construct the model with the respective initial parameters. As soon as all actors acknowledge the initialization, the Initiator instructs them to start the training phase, and then terminates itself.

3.7.2. Training Phase

The training phase is started by the Initiator after the actors have been initialized and comprises the actual training procedure of the DFL system. This includes training the local model, exchanging model updates, and aggregating the received model updates. To perform these steps, the following methods must be implemented by the respective learning strategy class:

- `fitLocal()`: Train the local model on local data.
- `broadcast()`: Send the model update to neighboring actors.
- `aggregate()`: Incorporate received model updates into the local model parameters.

The method `performTraining()` then orchestrates the iterative execution of these methods for the specified number of epochs. The `IDFLStrategy` interface class provides a basic implementation of the `performTraining()` method, which can be overridden in the respective learning strategy class to match the individual requirements.

3.7.3. Evaluation Phase

In the evaluation phase, the evaluation metrics are obtained using the current state of the model parameters. Note that the evaluation can take place after completion of the training phase, as well as in-between communication rounds during the training phase, depending on the implementation of the learning strategy class. MoDeFL offers two types of evaluation: Local evaluation and neighbor evaluation. With local evaluation (the `evaluate()` method in `IDFLStrategy`), the model of an actor is evaluated on its own local dataset. With neighbor evaluation (the `evaluateNeighbors()` method in `IDFLStrategy`), on the other hand, the actor sends its model parameters to the neighboring actors, requesting them for evaluation. The neighboring actors then evaluate these model parameters using their respective dataset and return the resulting evaluation metrics to the requesting actor. These evaluation metrics are eventually averaged, representing the evaluation results on foreign data.

3.8. Configuration

MoDeFL aims to be fully configurable to facilitate experimentation. Therefore, both the Initiator and the actors can be configured via command line arguments as well as through a dedicated configuration file, described in Section 3.8.2. Besides these optional configurations, which return to a default option when unspecified, the Initiator must be supplied with the respective network configuration (i.e., network addresses of actors and network links between actors), as outlined in Section 3.8.1.

3.8.1. Connectivity

To communicate with the actors, the Initiator requires knowledge about their network addresses. Since these addresses cannot be inferred automatically, we have to provide the Initiator with a list of the network addresses from the actors. This is done through a text file that lists the addresses line by line. In addition to the addresses, the Initiator requires knowledge about the network topology of the actors, as the Initiator is responsible to inform the actors about their neighboring actors. Therefore, we provide the Initiator with an adjacency matrix that defines which actors are connected to each other through binary values at the respective position in the matrix. The rows and columns of the adjacency matrix must follow the same order as the address file.

3.8.2. Configuration File

MoDeFL allows numerous configurations regarding the dataset, the local and global learning process, and general characteristics of the system through a dedicated configuration file in json format. The path to this file is provided as command line argument `--config=<path_to_config>` when starting the Initiator or the actors. Table 3.1 lists the configurations that are in common between the Initiator and the actors; Table 3.3

Argument	Default value	Description
`seed`	13	Random seed for reproducibility
`num_threads_server`	# logical CPUs	Number of threads for the gRPC service
`log_level`	`DEBUG`	Logging level for the console output

Table 3.1.: Common configurations

Argument	Default value	Description
`log_performance_flag`	`True`	Flag indicating whether to log training and validation metrics or not
`log_communication_flag`	`True`	Flag indicating whether to log the data volume transferred between actors
`log_dir`	`./log`	Path to the directory for storing log files

Table 3.2.: Actor configurations

shows the configurations for the Initiator; and Table 3.2 outlines the actor-specific configurations. All tables contain the name of the configuration, its default option, and a brief description. Note that most of the Initiator's configurations can also be specified in the actor's configuration file, but these configurations are overwritten during the initialization phase when the Initiator transmits its configurations to the actors.

Argument	Default value	Description
dataset_id	Mnist	Dataset to be used for training
partitioning_scheme	ROUND_ROBIN	Scheme for partitioning the dataset
partitioning_alpha	2.5	Common value for the concentration parameters of the Dirichlet partitioning scheme
addr_file	./addr.txt	Path to the address file containing the actor addresses
adj_file	./adj.txt	Path to the adjacency matrix file specifying the network topology
num_fed_epochs	5	Number of communication rounds to be performed
num_local_epochs	1	Number of local training epochs in every communication round
lr	Model dependent	Learning rate applied for local training at the actors
lr_global	Model dependent	Global learning rate considered in certain aggregation methods
learning_type	DFLv1	Learning strategy to perform
sync_strategy	ONE_FROM_EACH	Synchronization strategy for retrieving the model updates from the market
sync_strat_percentage	0.5	Percentage attribute for synchronization strategies
sync_strat_amount	2	Amount attribute for synchronization strategies
sync_strat_timeout	3	Timeout in seconds considered in certain synchronization strategies
sync_strat_allowempty	False	Flag indicating whether to count an empty update in the synchronization strategy or not
compression_type	NoneType	Compression method applied to the model updates
compression_k	100	Amount of non-zero values for sparsification methods
compression_percentage	0.2	Percentage of non-zero values for sparsification methods
compression_precision	8	Precision in bits for quantization methods
pdp_strategy	NoneStrategy	Strategy for partial device participation
pdp_k	2	Number of neighboring actors for partial device participation

Table 3.3.: Initiator configurations

3.9. Conclusions

We have explained the design of the individual system modules in MoDeFL. Furthermore, we have discussed possibilities for expanding the individual DFL components by additional strategies, demonstrating the extensibility of the modules in MoDeFL. We have also described the execution of the training and outlined the configuration options. By clearly separating individual DFL components into fully configurable modules, MoDeFL facilitates the experimental comparison of DFL techniques. For this reason, and due to the simple integration of additional techniques, MoDeFL fosters experimentation with novel DFL approaches. MoDeFL already implements several common baselines for comparing experiments, which can be conveniently selected through the configuration. Besides the integration of additional DFL techniques, which constitutes an ongoing task, future work involves further simplification of the configuration options. Additionally, MoDeFL would benefit from clear definitions of the resources available to each module. This will clarify the general context for a specific module and, thus, further simplify the integration of novel techniques.

4. Communication Efficiency in Decentralized Federated Learning: A Survey

4.1. Introduction

Although DFL tries to eliminate the communication bottleneck at the central node by removing the server and outsourcing its responsibilities to the individual devices (actors), communication bottlenecks can still occur at the actors themselves. In the simplest DFL strategy, the network consists of fully-connected actors, where each actor sends its model update to every other actor. Because every actor must receive updates from all peers, the communication bottleneck moves from the CFL server to each actor. Hence, just eliminating the server does not suffice to resolve the communication bottleneck. Nevertheless, we can alleviate the communication bottleneck in DFL by reducing communication cost through dedicated configurations of model aggregation techniques, synchronization methods, and network topologies. In addition to these general DFL configurations, we can apply techniques such as compression, local computing, and partial device participation to further reduce communication overhead.

4.1.1. Related Surveys

The rapid growth of FL and the increasing number of respective publications gave rise to numerous surveys summarizing the advances. Although the majority of these surveys focus on CFL, significant efforts have also been devoted to reviewing DFL, both in addition to CFL [80]–[83] and exclusively [79], [84], [85]. In this context, the authors in [79] analyze DFL in terms of federated architectures, network topologies, communication mechanisms, security and privacy approaches, and key performance indicators. Furthermore, they explore existing optimization mechanisms of DFL and provide a list of trends, lessons learned, and open challenges. In survey [80], the authors elaborate on FL solutions that focus on network topologies for both CFL and DFL by analyzing their advantages and disadvantages, and discuss the remaining challenges and future work for applying FL in specific network topologies. Similarly, the authors of [81] survey CFL and DFL approaches and classify the existing work based on their network topology. They discuss the general techniques behind FL from the perspectives of theory and application, and frame the current application problems—also providing the existing attack scenarios and defense methods. In [82], the authors focus on synchronization strategies, consolidation methods, and network topologies, assessing and contrasting the efficacy and constraints of methodologies and providing insights for future progression. In [83], the authors introduce a systematic and detailed perspective on DFL, including iteration order, communication protocols, network topologies, paradigm proposals, and temporal

Survey	Model Aggregation (Section 4.2.1)	Synchronization Method (Section 4.2.2)	Network Topology (Section 4.2.3)
[79]	∼	✓	✓
[80]	×	✓	✓
[81]	✓	×	×
[82]	×	∼	✓
[83]	✓	∼	∼
[84]	×	×	∼
[85]	×	∼	∼
Ours	✓	✓	✓

(a) Communication-influencing Components

Survey	Compression (Section 4.3.1)	Local Computation (Section 4.3.2)	Partial Device Participation (Section 4.3.3)
[79]	∼	×	×
[80]	×	×	✓
[81]	✓	×	×
[82]	×	×	×
[83]	∼	×	✓
[84]	∼	×	✓
[85]	✓	×	∼
Ours	✓	✓	✓

(b) Communication Optimization Techniques

Table 4.1.: Coverage of communication aspects for DFL constituents in existing surveys on DFL; ×: does not comprise communication; ∼: mentions communication; ✓: elaborates on communication.

variability. Moreover, they propose categorizations within this context, and they discuss possible solutions and future research directions. The authors in [84] review existing approaches for traditional and blockchain-based DFL, identify emerging challenges, and discuss future research directions. In survey [85], the authors examine DFL approaches that try to optimize performance and efficiency in terms of memory usage, communication cost, convergence under data heterogeneity, and computational effort. In addition, they categorize the approaches based on their method to address system heterogeneity and data heterogeneity, and the authors provide some application scenarios of DFL.

Existing reviews on DFL cover a significant share of recent approaches from the literature. Analogous to DFL approaches, some survey papers focus on certain challenges such as security and privacy [79], [81], data heterogeneity [85], and system heterogeneity [85]. A fraction of surveys also discuss communication aspects for selected methodologies. However, to the best of our knowledge, the current literature does not comprise a review paper that specifically focuses on the communication cost in the DFL system. Table 4.1 provides an overview of the aspects that are addressed in terms of communication efficiency by the individual survey papers.

4.1.2. Contributions and Organization

In this survey, we revisit DFL with a special focus on the open research challenge of communication efficiency[1] and discuss the latest advances in this field. In contrast to existing work, we explicitly elaborate on the communication efficiency of key system constituents that characterize communication in the DFL infrastructure. Thus, we provide a guideline with important considerations for constructing a DFL system in bandwidth-limited environments. To our knowledge, the current literature does not include surveys that elaborate on the communication efficiency of the entire DFL system. An overview of existing surveys and their coverage of certain communication aspects is listed in Table 4.1. For the sake of focus and clarity, the scope of this survey was limited to non-blockchain approaches for horizontal DFL[2]. Following an extensive search through DFL literature, we selected 13 approaches based on their recency and relevance in terms of communication efficiency [29], [30], [37]–[39], [87]–[94].

Our key contributions are as follows:

- We review DFL approaches from the literature with a particular focus on communication cost.
- We identify three essential DFL components with a major impact on communication cost and discuss three popular techniques to optimize communication efficiency in DFL approaches.
- We establish taxonomies within the three DFL components and within the three optimization techniques, and unveil their impact on communication.
- We discuss the interplay of DFL methods in recent literature, reveal combinations that have already been addressed by research, and summarize the experiments of corresponding approaches.

We organize this survey in the following structure: In Section 4.2, we elaborate on different implementations of essential components in DFL based on recent literature and discuss their impact on communication efficiency; Section 4.3 reviews supplementary techniques to optimize communication cost in DFL and discusses their respective impact; Section 4.4 analyzes common DFL approaches regarding the composition of techniques from Section 4.2 and Section 4.3 to draw comparisons and to reveal combinations covered by research, and summarizes the experiments of these approaches to assess the communication efficiency; Section 4.5 summarizes the insights from the previous sections and discusses trade-offs between the challenge of communication efficiency and other challenges in DFL.

[1] Communication efficiency refers to the minimized utilization of communication resources (e.g., bandwidth, network links, etc.) required to maintain convergence, and is assessed based on communication frequency, information transfer volume, and related metrics.

[2] In horizontal DFL, the individual datasets at the actors share the same feature space [86].

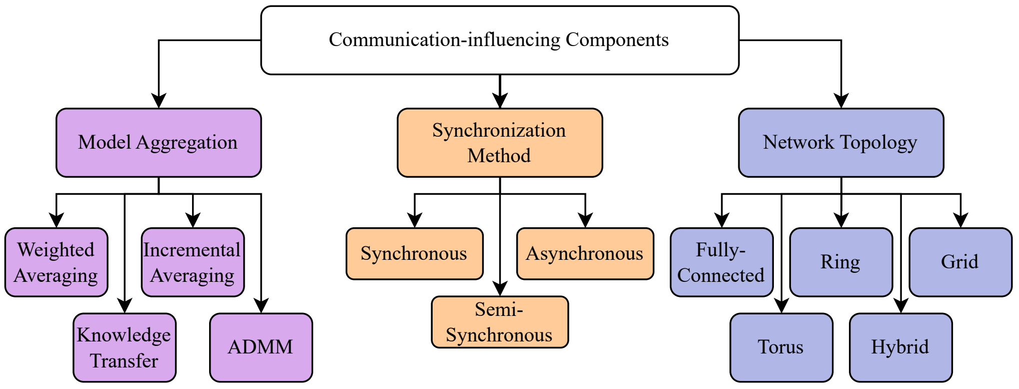


Figure 4.1.: Taxonomy with the communication-influencing essential components of DFL and their prevailing categories.

4.2. Communication-influencing Components

Communication efficiency in FL constitutes an ongoing research challenge. Albeit DFL addresses the communication bottleneck of CFL, there still exists a high demand for further improvements. In particular for systems with bandwidth limitations, exchanging a massive amount of model updates induces a major bottleneck in the DFL infrastructure. Therefore, reducing the communication cost brings substantial advancement in time performance.

In DFL, numerous system components determine the overall communication cost. Therefore, when calculating the communication overhead, we have to consider the question "*What* gets communicated *with whom* and *how often*?". This question is composed of three aspects: *What*, *with whom*, and *how often*. The "*what*" (i.e., the information transfer volume) is mainly influenced by the model aggregation technique (Section 4.2.1), as it decides the type of exchanged information (e.g., model parameters, gradient, etc.). The network topology (Section 4.2.3) affects the "*with whom*" (i.e., the network links for exchange) by specifying the neighborhood of an actor. Lastly, the question of "*how often*" (i.e., the communication frequency) is influenced by the synchronization method (Section 4.2.2), as it determines the system behavior in terms of waiting times and potential interruptions of communication rounds. Given these considerations, we identify the communication-influencing DFL components to be the Model Aggregation (Section 4.2.1), the Synchronization Method (Section 4.2.2), and the Network Topology (Section 4.2.3). In this section, we discuss the communication-influencing components in more detail by categorizing common implementations and summarizing the way and the extent to which they influence communication. The components are depicted in Figure 4.1 with their respective subcategories. Furthermore, Table 4.2 links the individual publications from common approaches to the corresponding subcategory of their implementation.

Reference	Model Aggregation	Synchronization Method	Network Topology
[87]	wAvg	Synchronous	Fully-connected, Ring, Torus
[29]	KT/KD	Synchronous	Fully-connected
[88]	wAvg	Synchronous	Random
[89]	wAvg	Synchronous	Fully-connected
[90]	wAvg	Synchronous	Fully-connected, Random, Other
[91]	wAvg	Synchronous	N/A
[39]	ADMM	Semi-synchronous	Random
[92]	wAvg	Synchronous	Fully-connected, Ring
[93]	wAvg	Synchronous	Torus
[37]	wAvg	Asynchronous	Other
[94]	incAvg	Synchronous	Other
[38]	wAvg	Asynchronous	Random
[30]	wAvg	Synchronous	Grid, Hybrid

Table 4.2.: Publications discussed in this survey ("Reference") with the corresponding categories of the communication-influencing components: Model Aggregation ("wAvg": weighted averaging; "incAvg": incremental averaging; "ADMM": alternating direction method of multipliers; "KT/KD": knowledge transfer/distillation), Synchronization Method, and Network Topology.

4.2.1. Model Aggregation

The model aggregation scheme is a central part of DFL, as it defines the method for incorporating received model parameter updates into the training process. As a core part of DFL, it plays an important role in communication efficiency, since it determines the type of information that actors need to exchange. In other words, it specifies the content of a model parameter update. Such an update could, for instance, contain one (or several) complete set(s) of model parameters, a gradient to previous models, or reduced model parameters, possibly combined with additional information about the model state. For that reason, the selection of an appropriate model aggregation scheme is crucial to achieve communication efficiency, as well as to ensure the convergence of the overall training.

Based on DFL literature, we focus on four general categories of model aggregation that are prominent in DFL: (1) weighted model averaging (wAvg), (2) incremental model averaging (incAvg), (3) alternating direction method of multipliers (ADMM), and (4) knowledge transfer and distillation. The authors in [83] introduce two learning paradigms that affect model aggregation: `Continual` and `Aggregate`. In `Continual` (also called continual federated learning or incremental federated learning), the actor resumes training directly on top of the previous actor's model parameters. In the `Aggregate` paradigm, on the other hand, the actor first aggregates several received model updates and then progresses training based on the aggregated model parameters. Since the `Aggregate` class covers a broad spectrum of aggregation techniques, we further categorize the model aggregation methods into sub-classes. In our context, incAvg falls into the `Continual` paradigm while the other three categories belong to the `Aggregate` paradigm.

wAvg The most popular class of aggregation schemes in DFL is the class of wAvg aggregation. As the name already suggests, strategies that follow the concept of wAvg implement an average over the set of received model parameter updates to obtain an aggregated state of parameter update in each round. Within this averaging process, neighbor-specific weights determine the relative influence of the individual contributions received from neighboring actors. Thanks to the ability to aggregate multiple updates, we can parallelize the training of the individual actors. Equation 4.1 shows the general form of an aggregation step for wAvg schemes in epoch k at actor i. To obtain the aggregated model $m_{k+1,i}$, actor i averages the model parameter updates $\Delta_{k,j}$ of its neighboring actors $j \in \mathcal{N}_i$, weighted by their respective mixing weight $W_{i,j}$. Note that with this notation, the set of neighboring actors $\mathcal{N}_i$ also contains the actor $i \in \mathcal{N}_i$ and that the corresponding mixing weights sum up to $\sum_{j \in \mathcal{N}_i} W_{i,j} = 1$. The model parameter update $\Delta_{k,j}$ refers to actor j's locally updated model parameters, for instance, by performing a local epoch of stochastic gradient descent (SGD). The simplicity of wAvg schemes and their suitability for parameter aggregation led to the development of numerous variants of wAvg-based aggregation schemes in DFL.

$$m_{k+1,i} = \sum_{j \in \mathcal{N}_i} (W_{i,j} \Delta_{k,j}) \tag{4.1}$$

The wAvg-based variants for DFL differ in three aspects: (1) the mixing weights $W_{i,j}$, (2) the set of neighbors' model updates considered for aggregation $\mathcal{N}_i^* \subseteq \mathcal{N}_i$, and (3) the content of a transmitted model update, consisting of the locally updated parameters $\Delta_{k,j}$ and potential supplementary information. We summarize the individual variants of common approaches regarding these three aspects in Table 4.3.

The **mixing weights** determine the impact of the individual model updates in the aggregation. The simplest strategy is to specify the mixing weights equally among all actors, giving each actor the same magnitude of impact [30], [87], [89]. A slight variation of this is to assign an individual weight to the aggregating actor while uniformly distributing the weights of the neighboring actors, thus adjusting the influence of the local model update separately [88]. Another strategy sets the mixing weights of the individual actors proportional to their respective data cardinality, prioritizing the model updates from actors with more data [93]. The choice of mixing weights does not directly influence communication in DFL; however, strategies that accelerate convergence, for example by setting the mixing weights according to network properties [91], [92] or dynamically adjusting them during training to capture time-varying connectivity [38] or model-specific behavior [37], [90], may reduce the number of required communication rounds.

The **selection of a subset of model updates** $\mathcal{N}_i^*$ after they have been received from the neighbors (i.e., model update selection) is rarely implemented in DFL, as it only brings a small reduction in computational effort and usually the contribution of all neighbors is valuable to some extent. However, the authors in [37] maintain a cache of model parameter updates from previous epochs and include these updates in the aggregation. They employ a model update selection technique based on reinforcement learning to prevent the consideration of stale updates. Most DFL approaches, however, leverage the optimization technique of partial device participation (Section 4.3.3) to reduce the amount of model updates involved in training. In contrast to the model update selection, partial device participation avoids the redundant transmission of model updates by determining the subset of neighboring actors before the transmission step. Hence, an actor transmits its model update only to a selected subset of neighboring actors. Note that both the selection of model updates and the technique of partial device participation can also be expressed in terms of the mixing weights by setting the respective weight to 0. Since the selection of model updates occurs after they have been received, this does not directly affect the communication efficiency.

The **content of a model update**, on the other hand, influences the communication cost in DFL directly, as it determines the size of the transmitted message. The simplest and most prominent approach is the transmission of the model parameters [30], [37], [88]–[90], [92], [93]. Other approaches exchange differential model parameters (i.e., model deltas) among the neighbors that are based on a common reference [87], [91], resulting in a generally smaller value range for the elements and, hence, yielding either a reduction in the communicated size or an increase in precision at the same communication cost. In [38], the authors include a scalar weight (i.e., the push-sum weight [95]) in addition to the model parameters with each model update, which is then leveraged to de-bias the local model. Beyond the model update, they communicate the training loss and the local iteration index to all available neighbors—regardless of the neighbors selected by partial device participation (see Section 4.3.3)—to support the neighbor selection in their implementation of partial device participation. The content of the model update directly influences the communication cost, since it determines the size of the communicated model update. Note that the communication between actors is not necessarily limited to the model updates, as some DFL approaches communicate additional information, either in the initialization phase [88], [90], [93] or in each communication round [30], [38], [89], [92] as detailed in Table 4.4. Also note that in the approaches of references [88], [89], [92], the additional information is communicated between the actors and a lightweight coordinator[3] to support their advanced training strategies from a central perspective. However, since the lightweight coordinators are not directly engaged in model aggregation, these approaches can still be considered decentralized, even though they incorporate a central node.

incAvg For the case of incAvg aggregation, the model parameters are passed through the network sequentially, with each actor locally optimizing the received parameters before forwarding the enhanced model parameters to the next actor. Thus, the model parameters travel through the network and undergo further optimization with each visit of an actor. This type of traveling through the network is also known as a random walk. During training, various states of the global model parameters are available in the network, since the individual actors only receive the global state when it is their turn to perform the local optimization step. In the local optimization step, the actor trains the received model using its private data before sending the updated model parameters to the next actor. The general formula for such a local optimization step on received model parameters m_p using SGD at actor i is given in Equation 4.2, where η is the learning rate, $f_i(\cdot,\cdot)$ is the loss function of actor i, x_i represents the private data on actor i, and ∇ is the differential operator.

When compared to DFL with wAvg, the incAvg strategy presents a different scenario.

[3] A lightweight coordinator is a central node that does not perform model aggregation.

Reference	Mixing Weights	Update Selection	Update Content
[87]	Equal	None	Estimated model delta
[88]	Equal (others)	None (PDP)	Model params.
[89]	Equal (both)	None (PDP)	Model params.
[90]	Trust-score (dynamic)	None	Model params.
[91]	Network topology	None	Model delta
[92]	Degree (global)	None (PDP)	Model params.
[93]	Data size	None	Model params.
[37]	Staleness (dynamic)	RL-based (cached)	Model params.
[38]	Degree	None (PDP)	Model params., scalar
[30]	Equal	None (PDP)	Model params.

Table 4.3.: Strategies of three fundamental aspects of wAvg-based model aggregation: Basis for the specification of Mixing Weights ("both": a total of two model updates; "others": different for the aggregating actor; "dynamic": dynamic re-weighting every round; "global": consideration on a global scope instead of the neighborhood), technique to perform Update Selection ("PDP": implementing Partial Device Participation (see Section 4.3.3) to select a subset of neighboring actors to consider; "cached": selection of updates from a cache; RL: reinforcement learning), and the Update Content ("params.": parameters) that is exchanged between the actors (model delta: difference in model parameters compared to a common reference).

Reference	Dir.	Additional Communication
[88]	$\rightharpoonup$	Matching decompositions, seed
[90]	$\rightleftharpoons$	Data size, data variance, connectivity
[93]	$\rightleftharpoons$	Data size

(a) Once for initialization.

Reference	Dir.	Additional Communication
[89]	$\rightharpoonup$	Seed, mixing weights matrix, round index
	$\leftharpoondown$	Bandwidth, loss, accuracy
[92]	$\rightharpoonup$	Set of neighbors, compression ratio
	$\leftharpoondown$	Consensus distance, communication capacity
[38]	$\rightleftharpoons$	Training loss, local iteration index
[30]	$\rightleftharpoons$	Parameters of last layer

(b) Every communication round.

Table 4.4.: Communicated information in addition to the model update content from Table 4.3; "Dir." indicates the direction of information transmission ($\rightharpoonup$ from coordinator to actors; $\leftharpoondown$ from actors to coordinator; $\rightleftharpoons$ among actors).

This is because it lacks computational parallelism, as only the currently visited actor performs local training at any given time. The class of incAvg is thus closely related to the field of continual learning and, consequently, faces many similar challenges as continual learning (e.g., catastrophic forgetting) [96]. Although the absence of computational parallelism increases the total execution time, incAvg aggregation drastically reduces the communication cost, as only one actor transmits the model parameters to another actor in each round. In this context, Gupta *et al.* [94] propose a novel strategy for selecting the path through the network to achieve the best possible accuracy with the least possible communication cost. More precisely, they consider all possible paths within a distance threshold to the shortest path in the network and select the path that offers the highest training accuracy. To do so, the connectivity and bandwidths of the entire network must be known. In their experiments, the authors show a significant reduction in communication cost while achieving similar accuracy as their baselines. In general, incremental training of the collaborative model is a powerful method to significantly reduce communication cost, but it is not suitable for time-critical applications.

$$m_i = m_p - \eta \nabla f_i(m_p, x_i) \tag{4.2}$$

ADMM ADMM is an algorithm to solve distributed convex optimization problems by decomposing the original problem into smaller sub-problems. First introduced in the 1970s [97], this method combines the benefits of dual decomposition and Lagrangian methods [98], thus allowing for a high degree of parallelization of the problem with numerical stability [81]. Given its suitability for decentralized optimization, ADMM was adopted by DFL. The general iterative updates for ADMM in a decentralized manner are provided in reference [99], as written in Equation 4.3 from the perspective of actor i. Here, $f_i(\cdot,\cdot)$ denotes the local objective function, c is the penalty parameter from the augmented Lagrangian, $\mathcal{N}_i$ is the set of neighbors, m_i^k is the local solution at iteration k, and λ_i^{k+1} are the local Lagrange multipliers. Similarly to Equation 4.2 for incAvg, x_i represents the private data of actor i and ∇ is the differential operator. Note that the neighboring actors exchange their model updates $\{m_j^k, \forall j \in \mathcal{N}_i\}$ (i.e., the local solutions) beforehand, as they are required for the update step. Also note that both the calculations of m_i^{k+1} and λ_i^{k+1} rely only on local information from the respective actor and the model updates of its neighbors, which means that the transmission of solely the model updates is sufficient. In [39], the authors adopt ADMM to DFL in combination with techniques of compression (Section 4.3.1), local computation (Section 4.3.2), and partial device participation (Section 4.3.3). To reduce the computational burden of ADMM, they implement the inexact alternating direction method of multipliers (iADM) [100]. With regard to communication, leveraging ADMM for model aggregation in DFL generates the equivalent cost as wAvg in each round. This results from the fact that the local solutions m_i^k, which are exchanged with neighboring actors, have the same dimension as the model parameters. Nevertheless, ADMM can solve non-smooth optimization problems effectively and offers a high degree of scalability.

$$\begin{aligned}
\lambda_i^{k+1} &= \lambda_i^k + c\left(|\mathcal{N}_i| m_i^k - \sum_{j\in\mathcal{N}_i} m_j^k\right) \\
m_i^{k+1} &= \Big(\nabla f_i(m_i^k, x_i) + 2c|\mathcal{N}_i| I\Big)^{-1}\left(c|\mathcal{N}_i| m_i^k + c\sum_{j\in\mathcal{N}_i} m_j^k - \lambda_i^{k+1}\right)
\end{aligned} \tag{4.3}$$

Knowledge Transfer and Knowledge Distillation The last class of model aggregation techniques that we discuss in this section is based on knowledge transfer and knowledge distillation. Knowledge distillation (also referred to as model distillation) is widely used as an effective technique for transferring knowledge from a large, powerful neural network (teacher model) to a smaller network (student model) by mimicking the teacher's behavior [101], [102]. Based on this idea of knowledge distillation, Zhang *et al.* [103] propose a strategy that enables multiple students with comparable models to learn to solve the task concurrently and collaboratively, which is called mutual learning. To achieve this, each student optimizes both the local loss and the loss from knowledge distillation with other students. As a result, students engage in collaborative training, eliminating the teacher role. In [29], the authors adopt the idea of mutual learning into DFL to address the problem of client drift[4]. Their approach (named DefKT) distinguishes between two roles for the participating devices: Sender and receiver. The task of a sender is to update its local model and transmit the resulting model parameters to its assigned receiver device. Subsequently, the receiver device incorporates the received model parameters into its local model by transferring the knowledge between these two models. This knowledge transfer is performed iteratively on batches of data. Equation 4.4 presents the updates of both the local model of the sender m_s and the received model m_r in the l-th iteration of knowledge transfer. Here, η_s and η_r denote the learning rates of the knowledge transfer update, B_l denotes the l-th data batch, and $P_{r,l}$ and $P_{s,l}$ are the soft predictions (i.e., the outputs of the model) obtained on the data batch B_l under the respective model m_s or m_r. The loss functions $\text{Loss}_s(\cdot)$ and $\text{Loss}_r(\cdot)$ consist of two additive terms: (1) The general, task-specific loss function evaluated on the soft predictions $P_{r,l}$ and $P_{s,l}$, and (2) the Kullback-Leibler divergence that quantifies the match of the two sets of soft predictions. More detailed information on these loss functions can be found in reference [29]. It is worth mentioning that, apart from iterating through the data batches, DefKT employs three different quantities of iterations: (1) The global number of epochs (i.e., communication rounds), (2) the number of local updating steps, and (3) the amount of training passes for the knowledge transfer process, where each training pass iterates through all data batches of the respective dataset at the actor. In terms of communication overhead, DefKT restricts the transmission of model updates to a group of senders and an equal-sized group of receivers in each communication round, both form subsets of the entire community of devices. Consequently, they drastically limit the number of activated communication links. As this also relates to the optimization technique of partial device participation, we further elaborate on the limitation of activated communication links in Section 4.3.3. The content of a model update in DefKT comprises the full set of model parameters. Therefore, the communicated size per model update is comparable to the general case of the other model aggregation strategies. Nevertheless, the technique of knowledge transfer and the additional configurable parameters in DefKT (e.g., the number of training passes of the knowledge transfer process) offer further possibilities to increase communication efficiency by speeding up the convergence.

[4]Client drift denotes the phenomenon when collaborating participants train in different directions due to data heterogeneity.

$$
\begin{aligned}
m_s &= m_s - \eta_s \frac{\partial \mathrm{Loss}_s(m_s, B_l, P_{r,l})}{\partial m_s} \\
m_r &= m_r - \eta_r \frac{\partial \mathrm{Loss}_r(m_r, B_l, P_{s,l})}{\partial m_r}
\end{aligned} \tag{4.4}
$$

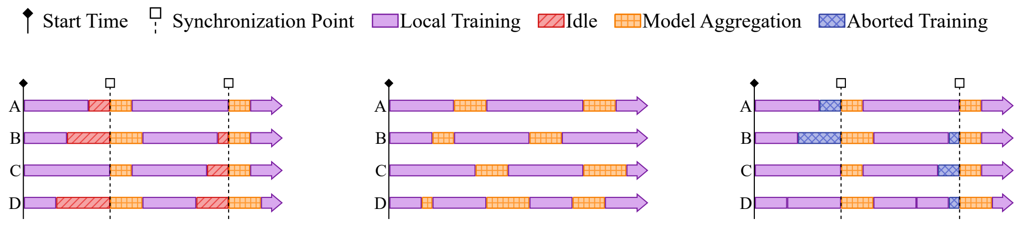
With all of the model aggregation categories, the overall communication cost is indirectly influenced by the convergence rate. Depending on the exact optimization problem and the dataset, certain model aggregation strategies increase the convergence rate, thus reducing the number of communication rounds required to reach a certain accuracy. Since the number of communication rounds determines how often we perform the exchange of model updates, it constitutes a major driver for the overall communication cost. Apart from this, the model aggregation schemes also exhibit a direct effect on the communication cost, as discussed in previous paragraphs. Important to note here is that while the model aggregation class (wAvg, incAvg, ADMM, or Knowledge Transfer) defines the general framework for the aggregation strategy and, therefore, determines its baseline communication cost, specific implementations within a class may introduce additional communication overhead. For example, a wAvg scheme that optimizes the mixing weights based on statistics about the local training steps of the actors increases the communication cost by including these statistics in the content of the transmitted model update. To summarize, we have described the state-of-the-art categories of model aggregation schemes to provide insight into the general framework in which their implementations operate, and we have discussed their general implications for communication efficiency, outlining key differences in communication between the respective aggregation types.

4.2.2. Synchronization Method

The synchronization method determines the behavior of the actors when exchanging model updates. Essentially, it defines whether the actors wait for the model updates from the other actors or continue the training process directly after sharing their model updates. Since the respective synchronization method is an instrumental aspect of the DFL system, it affects both the performance of the overall learning process and the general operability of other techniques in the DFL system. Furthermore, different synchronization methods entail different challenges, such as the challenge of stale model updates and the challenge of stragglers. In terms of communication cost, the synchronization method strongly influences the number of communicated model updates.

Following existing literature (see e.g., [79], [80], [82]), we distinguish between the categories of synchronous, asynchronous, and semi-synchronous DFL, as they provide a solid separation regarding their main differences. We elaborate on the characteristics of these three categories, put them in contrast, and discuss their impact on the communication cost. An exemplary execution timeline of four actors in the DFL system is shown in Figure 4.2 to visualize and compare the respective behavior of the three synchronization categories.

In **synchronous** DFL, all actors synchronize with each other directly after they perform local training. As a result, the model updates that are exchanged among the actors all have the same level of progress, which simplifies the aggregation process. Due to its



(a) Synchronous (b) Asynchronous (c) Semi-synchronous

Figure 4.2.: Exemplary execution timeline on four actors (A/B/C/D) for the three types of synchronization methods. The synchronous scheme (Figure 4.2a) involves idle times; the semi-synchronous scheme (Figure 4.2c) includes aborted trainings; and the asynchronous scheme (Figure 4.2b) shows increased aggregation times due to varying situations.

simplicity in aggregation and, thus, also in convergence proofs, the synchronous type is the most popular category among the synchronization methods [29], [30], [87]–[94]. However, heterogeneous actors cause idle times in the learning process, since all actors have to wait for the slowest actor in the system (i.e., the problem of stragglers). As indicated in Figure 4.2a, faster actors wait for all other actors to complete local training (purple bar) in an idle state (red bar). Only when all actors have completed the local training will they exchange the model updates, aggregate them (orange bar), and continue with the next iteration (i.e., the next communication round). The point in time when all actors have completed the local training is called the synchronization point. In synchronous DFL, the communication is limited to a single exchange of model updates per communication round, which occurs at the synchronization point. As the actors wait for each other, they perform only one local training in each communication round, and, hence, they share only a single model update with their neighbors. Note that even if we perform multiple local epochs in the local training step using the optimization technique of local computation (see Section 4.5), we obtain only one overall model update that covers these local epochs. Thus, each actor is strictly limited to sending only one model update to its neighbors in each communication round.

In contrast to synchronous DFL, actors in **asynchronous** DFL do not wait for other actors to complete their training step. Instead, an actor directly sends the model update to its neighboring actors after finishing the local training and continues with the subsequent aggregation process. Hence, this model aggregation process does not wait for any model updates from the neighboring actors but only involves the model updates that are available at that time. To enable this asynchronous communication, every actor permanently listens for incoming model updates and stores them in a buffer until it performs the model aggregation. The asynchronous scheme addresses the problem of stragglers encountered in the synchronous scheme, yet it introduces additional challenges. First, the model aggregation method must allow for an arbitrary number of model updates, since we cannot determine the time required for an actor to perform a model update in advance. Second, the model aggregation must allow for multiple model updates from the same neighboring actor. Finally, we have to consider the challenge of stale model updates which adversely affect the optimization. Figure 4.2b depicts the execution timeline for the asynchronous scheme, clearly showing the elimination of idle times that occur in the synchronous case. Note that we encode an overall increase in the duration of model aggregation in Figure 4.2b compared to synchronous DFL, which is due to its higher complexity and the additional check for staleness of the model updates. The actors in asynchronous DFL directly continue with the consecutive local training step and the transmission of the successive model update after performing the model aggregation. Model updates are therefore communicated at a higher frequency among the actors than it is the case with the synchronous scheme (unless the overhead in the duration for model aggregation exceeds the idle times from synchronous DFL). As a result, faster actors share more model updates than slower actors within the same time, causing asynchronous DFL to involve more communication than synchronous DFL. Note, however, that asynchronous communication has the advantage of sending model updates at different times, which prevents the actor from being flooded by a massive amount of updates at once. Among the publications discussed in the present survey, two approaches employ the asynchronous synchronization method [37], [38]. Similar to the synchronous scheme, the concept of the asynchronous scheme is also strictly defined. Nevertheless, differences in the approaches exist in their handling of surrounding challenges, such as the increase in complexity of convergence and the problem of staleness. In this context, the authors of [38] provide a convergence analysis of their approach under the heterogeneous scenario, assuming strongly convex functions. Liu *et al.* [37], on the other hand, propose a dynamic, staleness-aware model aggregation method to address the problem of stale model updates, which is further discussed in Section 4.2.1.

The **semi-synchronous** method combines characteristics from the synchronous and the asynchronous scheme. Similarly to the synchronous method, the semi-synchronous scheme involves synchronization points on which all actors synchronize with each other (i.e., perform model aggregation). In between two synchronization points, the actors do not wait for each other after completing local training. Instead, they directly start the next iteration of local training. Hence, there are no idle times with the semi-synchronous method, just like in the asynchronous scheme. The key characteristic of semi-synchronous DFL is that the actors abort the local training step when reaching a synchronization point. Aborted trainings are colored in blue in Figure 4.2c. Although aborting the local training step wastes computational effort, the benefit behind this concept becomes apparent when multiple local training steps can be completed within one synchronization period (i.e., the time between two synchronization points). As shown in Figure 4.2c, actor D successfully performs multiple local training steps in-between synchronization points, leveraging available computing resources that would be idle in synchronous DFL. Variations of the semi-synchronous method differ in the determination of synchronization points. In Figure 4.2c, the synchronization points are determined by the time required by the slowest actor to complete one local training. However, the synchronization points can also depend, for instance, on a pre-specified duration, on the fastest actor, or on more sophisticated evaluations among the actors (e.g., a metric of divergence). Note the risk of confusing the semi-synchronous scheme with the synchronous scheme when the technique of local computation (see Section 4.3.2) with multiple local updates is involved. The main difference is that the semi-synchronous scheme describes the parallel execution of all participating actors, whereas local computation determines the number of local updates performed before the model parameters are aggregated with model updates from other actors—applicable with all synchronization methods. The semi-synchronous scheme exhibits similarities to both the synchronous and the asynchronous scheme. In terms of communication, the frequency of transmitted model updates almost resembles the asynchronous method, since there are no waiting times. The difference lies in the aborted training steps, which do not yield a model update. Whether semi-synchronous synchronization exhibits a higher or lower communication frequency than asynchronous DFL depends on both the heterogeneity of computing resources between the actors and the time required for the model aggregation step. The bandwidth bottleneck that arises in synchronous DFL due to receiving multiple updates all at once also exists in the semi-synchronous scheme. This is due to the common synchronization points, which lead to actors starting the training round simultaneously. As an implication, actors are completing the round and communicating the updates concurrently, given that they have similar computing power. However, this is not the case with heterogeneous computing resources at the actors. Semi-synchronous DFL particularly addresses the challenge of stragglers, since slow actors do not prevent the other actors from continuing their computations under the semi-synchronous scheme. Additionally, the semi-synchronous method benefits from joint synchronization points, reducing the complexity of convergence. In [39], the authors propose a variant of the semi-synchronous scheme that combines key characteristics from both synchronous and asynchronous DFL, called partial synchronization. Similarly to synchronous DFL, their variant comprises waiting times. However, the actors only wait for a pre-specified minimum amount of model updates. Slower actors are considered stragglers in that particular synchronization round and, hence, are not included in the model aggregation. As a result, the actors follow an asynchronous behavior throughout the synchronization periods. Due to the presence of partial synchronization points and asynchronous behavior, the method of partial synchronization belongs to the semi-synchronous synchronization methods.

To summarize, each of the three synchronization methods has considerable advantages but also entails some drawbacks. The synchronous synchronization method creates a more predictable environment regarding convergence of the overall learning process. Synchronous communication, however, does not address the problem of stragglers in the DFL system and is more susceptible to bandwidth bottlenecks. The asynchronous synchronization methods, on the other hand, address the challenge of stragglers by eliminating waiting times for the other actors. Instead of waiting for the others' model updates, an actor incorporates only available model updates and then directly continues its execution, which drastically increases the complexity of theoretical convergence and, thus, reduces the predictability of the overall learning process. Compared to synchronous DFL, the asynchronous method increases the amount of model updates transmitted between the actors while lowering the bandwidth bottleneck that arises from receiving multiple updates at the same time. The semi-synchronous synchronization method tries to combine the advantages of both the synchronous scheme and the asynchronous scheme by allowing for asynchronous behavior within a framework that enforces synchronization points to maintain the simplicity of model aggregation and convergence. This leads to trade-offs between the benefits and drawbacks, since underlying concepts are mutually exclusive. Therefore, variants of the semi-synchronous scheme try to optimize these trade-offs.

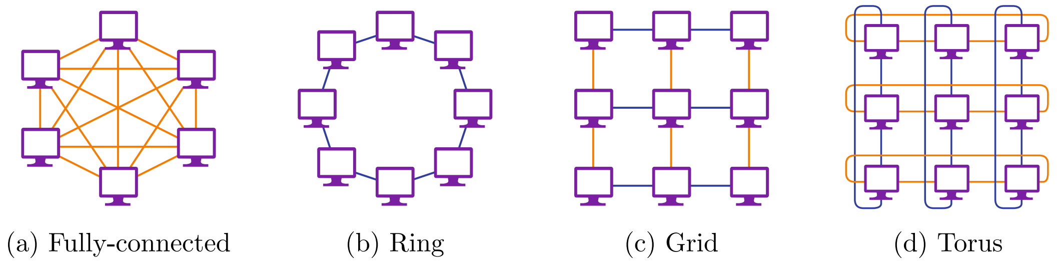


Figure 4.3.: Illustration of popular network topologies considered in DFL publications. In the Fully-connected topology (Figure 4.3a), all actors are interconnected; in the Ring topology (Figure 4.3b), each actor is connected to two neighboring actors, forming a ring; in the Grid topology (Figure 4.3c), connections form an evenly-spaced two-dimensional grid; and in the Torus topology (Figure 4.3d), the grid topology is extended by connecting the actors beyond the grid boundary to the other side.

4.2.3. Network Topology

As mentioned in Section 2.2, inter-node communication is a characterizing part of DFL. To facilitate communication in the first place, we require network connectivity between the actors. However, we cannot assume that all nodes are interconnected because of connectivity limitations (e.g., absence of network links between some actors) and contractual restrictions (e.g., no data contract[5] established between two parties). In this context, the network topology defines which actors are connected through a uni-directional or bi-directional network link. Similar to topologies in network science, we represent network topologies as adjacency matrices and visualize them as graphs. The most popular types of network topologies considered in DFL comprise the fully-connected mesh topology, the ring topology, the grid topology, and the torus topology, depicted in Figure 4.3. Due to the strict definition of these topologies and the necessity to consider arbitrary network structures in DFL, these network topologies are often combined to form hybrid variants.

In FL, we distinguish between two types of network connectivity: (1) The consistent connectivity arising from connectivity limitations and contractual limitations, which remains the same during the whole training process, and (2) the inconsistent (or variable) connectivity that changes over time due to node outages and the optimization technique of partial device participation (Section 4.3.3). Note that in this section, we cover the consistent connectivity, referred to as network topology[6]. Inconsistent connectivity is regarded with partial device participation in Section 4.3.3, given their common characteristic that actors do not receive model updates from all of their neighboring actors and the similar change in connection links. Although the active links are not selected by the partial device participation strategy but determined by inconsistent connectivity, the reception of model updates is similar in both cases.

[5]A data contract defines the terms for exchanging data (in our case model parameters) between two parties.

[6]In this survey, the term ‘network topology’ refers exclusively to consistent connectivity and, hence, does not encompass inconsistent connectivity, even though this is sometimes referred to as ’variable network topology’ in the literature.

Fully-connected Topology The fully-connected mesh topology—sometimes also referred to as complete graph topology—assumes that all actors are connected to each other actor in the network. Thus, with a total number of n actors, each actor maintains $n-1$ bi-directional network links (see Figure 4.3a). Works on DFL consider a fully-connected network topology [87], [89], [90], [92] either as the basis for their implementation or in their experimental setup. An important point to note is that a fully-connected network topology does not necessarily imply that every actor communicates with all other actors, since the variable connectivity (i.e., inconsistent connectivity) may further restrict the connectivity. In this context, the authors in [89] and in [92] employ techniques from partial device participation (Section 4.3.3) in their experiments, thus reducing the communication to a subset of selected actors. Without further restrictions on the connectivity, the model updates are transmitted from each actor to all other actors, resulting in the maximal possible amount of $n(n-1)$ transmissions per communication round. Also worth mentioning is that in addition to considering the fully-connected mesh topology in their experiments, the authors in [87] perform experiments with the ring topology and the torus topology, the authors in [90] conduct experiments with the chain topology and the random topology, and the authors in [92] include the ring topology in their experiments. Thereby, they provide a comparison of fully-connected mesh topologies in DFL with other topologies.

Ring Topology In a ring topology, each actor is connected to exactly two neighboring actors, forming a closed chain of connected nodes with a single continuous path through the network (see Figure 4.3b). Ring topologies emerge in DFL to reduce communication overhead, since the communication cost of DFL with a ring topology grows linearly with the number of participants. More specifically, a newly added actor places itself between two actors in the ring, resulting in one additional network link over which model parameters are exchanged. Thus, the number of model update transmissions in each communication round is given by $2n$. Koloskova *et al.* [87] perform experiments to compare the convergence of the fully-connected mesh topology, the ring topology, and the torus topology with $n \in \{9, 25, 64\}$ actors. Thereby, they show that such connectivity restrictions as in the ring topology cause only a mild negative impact on the convergence while reducing the order of communication growth. Furthermore, in [92], the authors consider two different methods for constructing the ring network, (1) randomly connecting the nodes to a ring topology or (2) constructing a consensus-aware ring greedily to maximize the consensus-distance between neighbors. To analyze the impact of the network topology, they additionally compare the performances of the ring topology and the fully-connected mesh topology in their experiments. In [30], the authors evaluate their proposed method using a hybrid topology called Ring of Cliques, in which 4 fully-connected cliques of 4 actors each are connected in a ring structure (16 actors in total).

Grid Topology Another characteristic network constellation is the grid topology. Here, the actors form an evenly-spaced two-dimensional grid connecting to their respective neighbors (see Figure 4.3c). Hence, inner nodes of the grid connect to 4 adjacent nodes, while edge nodes connect to only 3 nodes and corner nodes connect to 2 neighbors. Consequently, actors in a DFL infrastructure with grid topology exchange their model parameter updates with up to 4 neighbors. In a square grid constellation, for instance, one communication round comprises $4(n - \sqrt{n})$ transmissions of model updates. However, only a few DFL approaches consider such (artificially constructed) grid topologies due to their rarity in real-world deployments. Soltani *et al.* [30], for instance, perform experiments using the grid topology as well as the hybrid Ring of Cliques topology, with 16 actors each. When training with the grid topology, their results indicate slightly faster convergence in terms of communication traffic than for the Ring of Cliques. The difference, however, is so small that it does not allow for any general claims in this regard.

Torus Topology The torus topology is an extension of the grid topology in which the edge nodes additionally connect to the corresponding nodes on the opposite edge (see Figure 4.3d). As a result, all nodes have a degree of 4 (i.e., connect to 4 nodes). The torus topology is sometimes also referred to as a 2D ring, since connecting an edge node to the opposite edge forms a continuous path (i.e., ring) through the respective row or column of the grid. Since each actor is connected to exactly 4 neighboring actors, there are $4n$ model update transmissions in each communication round. In the experiments of [87], the comparison of performance on the fully-connected mesh topology, the ring topology, and the torus topology demonstrates only mild performance differences between these topologies, with the torus topology falling between the other two in terms of performance. Beyond employing the torus topology for experiments, Zong *et al.* [93] introduce a 2D-attention-based device placement algorithm to minimize the overall communication cost. Here, the algorithm arranges the actors within the torus schema intending to minimize both the transmission time and the communication latency.

Other and Hybrid Topologies Besides these popular network topologies, also other topologies like the chain topology (i.e., ring topology with one link removed) [90], the exponential graph topology (i.e., every actor is connected to $\lceil \log_2 n \rceil$ actors) [37], or a random graph (i.e., connections are randomly constructed, for instance with the Erdős–Rényi random graph model) [38], [39], [88], [90] are considered in the literature. Moreover, network topologies are sometimes combined into hybrid topologies, such as the "Ring of Cliques" topology in reference [30], which arranges so-called cliques (i.e., fully-connected groups of actors) in a ring topology. In general, hybrid topologies can combine arbitrary network topologies.

We have seen that network topologies highly influence the communication cost of a DFL system, as they determine which actors can communicate with each other. Thus, network topologies restrict the number of possible model update transmissions. Among the topologies discussed, communication is most restricted by the ring topology, followed by the grid topology and then the torus topology. The fully-connected topology does not restrict communication, as it allows every actor to communicate with all other actors. The majority of DFL approaches consider fully-connected mesh topologies and random graph topologies (see Table 4.2). However, it is important to note that various strategies exist for constructing random graph topologies. Therefore, a direct comparison of the "popularity" of random graph topologies with that of strictly defined topologies is not meaningful.

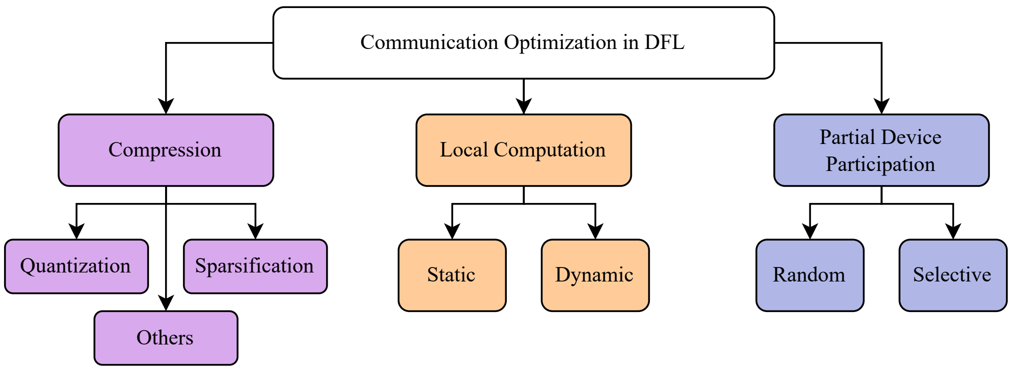


Figure 4.4.: Taxonomy with the most prominent techniques to optimize the communication of DFL and their key categories.

4.3. Communication Optimization Techniques

In addition to the essential components of DFL infrastructures discussed in Section 4.2, various optimization techniques can be applied to further enhance the communication efficiency. While these techniques are not strictly indispensable for a DFL system, they are of gread utility for reducing the communication cost. We identified three major categories of communication optimization techniques in DFL, which we discuss in the following subsections: Compression (Section 4.3.1), Local Computation (Section 4.3.2), and Partial Device Participation (Section 4.3.3). For each of these categories, we outline its origins, identify prominent classes, explain the general concept, and discuss the impact on communication cost with emerging challenges and trade-offs. Figure 4.4 illustrates the taxonomy and Table 4.5 links the underlying classes to common approaches from recent publications.

Many of the following techniques originated in CFL or in distributed ML, and have been adopted to DFL. However, since DFL features a different setting than CFL, these methods require adaptation to match the changes in system characteristics.

4.3.1. Compression

Model compression is a well-known concept for reducing the communication cost between the clients and the server in CFL systems. In this process, each client applies lossy compression to the model parameter updates before transmitting them to the central server for aggregation (downlink compression). Likewise, the server responds with a compressed version of the aggregated model parameters (uplink compression). Popular compression methods for this purpose include quantization methods and sparsification methods. However, the loss of information involved usually sacrifices training performance of the global model and, hence, creates a trade-off between compression ratio and training performance. Therefore, research groups continue to develop compression techniques that maximize the compression ratio while preserving good convergence rates.

In DFL systems, we can implement compression methods in a similar way as in CFL systems. The main difference here is that we compress the model parameter updates that are communicated between the individual actors. In contrast to compression in CFL, the

Reference	Compression	Local Computation	Partial Device Participation
[87]	Quantization, Sparsification	None	Random (not in experiments)
[29]	None	1, 10 Updates	Random 20%
[88]	None	Implicitly (PDP)	Selective (connectivity)
[89]	Sparsification	None	Selective (bandwidth), single
[90]	Sparsification	5 Updates	None
[91]	Quantization	4 Updates	None
[39]	1BCS	Random in $[5, 15]$	Random 20%, 50%, 80%
[92]	Sparsification	50 Updates	Selective (consensus)
[93]	None	None	None
[37]	Sparsification	Implicitly (PCP)	Random, single
[94]	None	1, 2, 5, 10 Updates	None
[38]	None	2 Updates	Selective (loss)
[30]	None	5 Updates	Selective (similarity), single

Table 4.5.: Publications discussed in this survey ("Reference") with the corresponding implementation categories of the communication optimization techniques: Compression ("1BCS": 1-bit compressed sensing [104]), Local Computation ("Implicitly (PDP)": the technique of Partial Device Participation causes devices to perform multiple consecutive local updates), and Partial Device Participation ("Random": participating devices are randomly selected; "Selective (`XX`)": participating devices are selected based on `XX`; "Single" / "`PP`%": only one / `PP`% neighboring devices participate in every round).

scattered nature of communication links in DFL introduces additional challenges in terms of maintaining convergence. Therefore, taking into account the system environment (e.g., network topology, synchronization method, etc.) is of high importance for implementing compression in DFL.

In the literature, two compression techniques dominate for the use with DFL: Quantization [87], [91] and sparsification [37], [89], [90], [92]. Both techniques belong to the class of lossy compression techniques, which use inexact approximations to compress data. Note that lossless compression techniques are generally not suitable for DFL as they require an expected structure in the data, which is not the case with model parameters or gradients. With lossy compression techniques, it is crucial to consider the impact on the model accuracy caused by the loss of information due to the entailed approximations. However, with a proper balance between model performance and system efficiency, it is possible to significantly reduce the size of the model parameters while maintaining reasonable accuracy [105].

Quantization In literature on DFL approaches, we encounter various quantization schemes. In general, quantization compresses data by reducing the number of bits required to represent the data. A simple but popular quantization scheme is randomized quantization. Implemented (among others) in reference [87], randomized quantization divides the data range into a set of evenly distributed quantization levels (i.e., bins) and then randomly rounds the data values to the adjacent lower or upper bin. Consequently, we reduce the data size—and thus reduce communication cost—by placing the bins on values that can be represented by a smaller quantity of bits. In addition to uniform quantization methods like randomized quantization, DFL also adopts non-uniform approaches. In [91], for instance, the authors apply the Lloyd-Max quantization algorithm and adjust the quantization levels adaptively to minimize the involved distortion. Thereby, they accounts for time-varying convergence rates and variable gradient distributions while providing convergence guarantees without convex loss assumptions. Besides reducing the data size, the authors in [106] incorporate quantization to achieve differential privacy guarantees, jointly addressing the challenge of communication efficiency and privacy concerns.

Sparsification Sparsification in distributed training (i.e., the exchange of sparse updates) has shown great potential to reduce communication cost without drastic consequences for the convergence [107]. Similarly, the concept of sparsification gains tremendous popularity in DFL to decrease the amount of transmitted parameters. Basically, sparsification generates a sparse representation of a matrix by removing (i.e., zeroing) a considerable number of entries. However, various strategies exist for sparsification in DFL, trying to optimize the trade-off between communication efficiency and information loss. In this regard, the authors in [92] implement the method of random sparsification (RandomK), that preserves k randomly selected entries, and optimizes it by determining actor-specific compression ratios. The semi-decentralized FL framework GossipFL [89], on the contrary, generates a random sparsification mask at every client based on a common seed received from the light-weight coordinator. Therefore, all model parameter updates communicated among the actors exhibit the same sparsity structure in each communication round. The sparsity masks are generated by drawing from a Bernoulli distribution with a probability of $\frac{1}{c}$, where c is the compression ratio specifying the fraction of non-zero elements after sparsification. Besides randomly constructing the sparsification mask, a highly popular method is TopK sparsification, which keeps only the k elements with the highest magnitude [107], [108]. In [90], the authors propose an improvement of TopK sparsification in DFL by adding a momentum to the residual. This additional momentum restricts the impact of the current gradient, resulting in a more stable training and improved performance. Another approach to retain convergence rate with sparsification is proposed by the authors in [37]. Their adaptive sparse training method minimizes the impact of sparsification on the loss function and simultaneously considers the imminent effect of pruned parameters on the training process.

In addition to quantization and sparsification methods, several other compression techniques exist which qualify for reducing communication cost in DFL. However, quantization and sparsification are the most popular strategies, as they have proven their suitability for optimization tasks. Also worth mentioning is the DFL approach in [39], which combines sparsification with quantization by using one-bit compressive sensing (1BCS) [104]. Although 1BCS itself belongs to the group of quantization techniques, its input data must be sparse, requiring the employment of sparsification methods. Therefore, the authors in [39] perform model training subject to a sparsity constraint before applying 1BCS to allow the transmission of one-bit information among the actors. Besides DFL approaches that explicitly implement compression techniques [37], [39], [87], [89]–[92], compression methods are typically orthogonal to other components of the system and, hence, can be integrated on top of the existing framework.

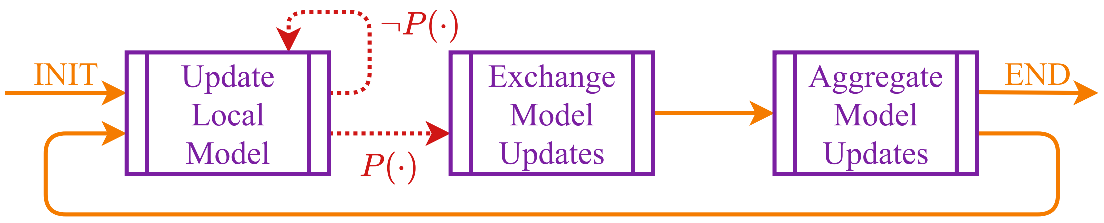


Figure 4.5.: High-level DFL process with Local Computation through the predicate function $P(\cdot)$; Performing consecutive local updates until $P(\cdot)$ evaluates positively.

4.3.2. Local Computation

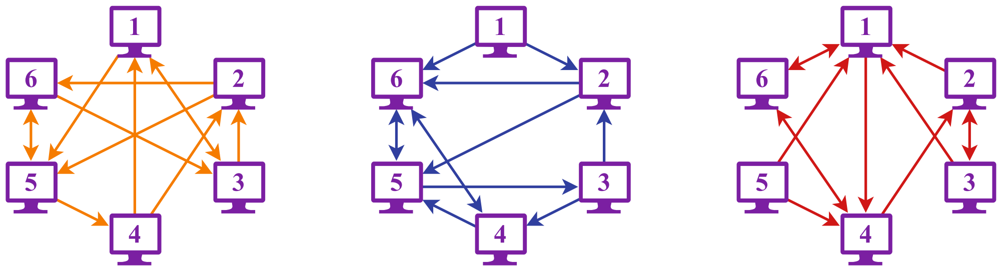
The concept of local computation (also known as local-updating SGD, intermittent communication, or federated averaging) is a simple yet powerful strategy to enhance communication efficiency in DFL systems. With local computation, multiple local updating steps are performed at the actor (instead of just one) before the updates to the model parameters are exchanged with the neighboring actors and the received updates are aggregated (synchronization step). Consequently, the actors only communicate after several local updates rather than after each update, reducing the communication by a constant factor. However, this intermittent synchronization introduces a trade-off between convergence rate and communication efficiency, as it gives the actors leeway to diverge. Alongside its numerous applications in CFL [4], [109], [110] and its implementation in the Federated Averaging algorithm [1], the local computation strategy shows success in general distributed optimization systems [111]–[113]. As a natural consequence, this concept is implemented by many DFL approaches to optimize communication efficiency [29], [30], [38], [39], [90]–[92], [94].

The implementations of local computation in distributed ML, CFL, and DFL follow the same concept, since all of them involve individual devices that perform computations locally and exchange updates to collaboratively approach a common goal. What changes between (and also within) these scenarios, however, is the general setting (e.g., connectivity, computing resources). Therefore, we should consider different behavior in terms of convergence, accuracy, and communication in the respective setting when adopting the concept of local computation.

Applications of local computation can be divided into two distinct categories, depending on whether the quantity of successive local updates is statically fixed or dynamically adapted. With static specification, each actor in the DFL system performs a constant number of τ local iterations per global synchronization. With dynamic adaptation, on the other hand, the point of aggregation is determined dynamically based on criteria such as divergence to a global reference model [114] and computing resources of the individual devices [115]. Figure 4.5 depicts the high-level DFL process including general local computation through a predicate function $P(\cdot)$ which is represented by the dotted red lines. Here, the result of $P(\cdot)$ indicates whether to perform another round of local updating or to proceed with the exchange and aggregation of model parameter updates (i.e.,, the synchronization step). The construction of $P(\cdot)$ allows to statically specify the amount of local updates τ (e.g., $P(t) : t \mod \tau = 0$, with current epoch t) as well as to define more sophisticated, dynamic synchronization rules. Parameter synchronization in a statically specified interval is often referred to as periodic synchronization.

Static In the context of DFL, the majority of approaches that implement local computation define static quantities of local updates. It is important to note, that the proper choice of the amount of local updates highly depends on the respective data properties and the system configuration. To excel the involved trade-off between convergence rate and communication efficiency, we thus have to consider (1) the heterogeneity of data partitions among actors to avoid concept drifts, (2) the computing resources of the individual actors to allow for fairly-balanced contributions, and (3) the general communication scenario (i.e., network topology (Section 4.2.3), synchronization method (Section 4.2.2), and partial device participation (Section 4.3.3)) to reduce the communication overhead while ensuring global convergence. By restricting the synchronization of model parameters to every τ rounds instead of every round, we reduce the communication cost by a factor of $1/\tau$. Note that the efficiency of local computation depends on the inter-play of the amount of local updates τ, the applied batch size, and the respective learning rate, as we can decrease the batch size and the learning rate to allow for an increase in τ. Within the scope of common approaches, the authors in [29], [30], [38], [90], [91] conduct experiments with a static quantity of up to 10 local updates. Furthermore, in [92], the authors perform 50 local training epochs with a batch size of only 32 data samples. In [94], on the other hand, they compare the accuracy with 1, 2, 5, and 10 local updates in their experiments, showing the importance of performing at least 2 local updates to accelerate convergence in their experimental setup. In the experiments of [39], the respective authors allow each actor to perform an individual amount of local training epochs, randomly drawn from $[5, 15]$. However, they do not elaborate on the impact of heterogeneous quantities of local updates.

Dynamic While, to the best of our knowledge, literature does not yet explicitly address dynamic strategies for local computation specifically in DFL (except with the help of a central node [41]), they are successful in distributed ML [114], [115]. Given the strong similarity of DFL to distributed ML in this regard, we can adopt many dynamic approaches for local computation from distributed ML directly into DFL. Nevertheless, two common approaches from recent literature implicitly employ a variable number of local training epochs [37], [88], thus belonging to the category of dynamic local computation. It is important to note that these approaches do not explicitly facilitate multiple local training epochs. Instead, other system components cause the repeated local training of the actors. More specifically, the authors in [88] incorporate a random selection of network links from partial device participation (Section 4.3.3), which could result in actors not exchanging model updates, eventually leading to consecutive local updates. In reference [37], on the other hand, each actor sends its model parameter update to one random neighbor in each round, which may result in some actors not receiving a model update and, hence, continuing the training on stale model parameters (i.e., performing a consecutive local training epoch).



(a) Epoch 1 (b) Epoch 2 (c) Epoch 3

Figure 4.6.: Exemplary Partial Device Participation with random selection of 40% (i.e., 2/5) neighboring actors over three epochs—drawing a new subset of activated links in each epoch (as illustrated in 4.6a, 4.6b, and 4.6c).

4.3.3. Partial Device Participation

Partial Device Participation in FL has already been implemented in 2016 when the Federated Averaging algorithm was introduced [1]. The goal of partial device participation, also known as client selection and neighbor selection, is to reduce the communication overhead by limiting communication of each actor to only a subset of neighboring actors in our DFL system. Thereby, this technique further restricts the network connectivity in the DFL system, since it only allows the transmission of model parameter updates over a fraction of network links. However, in contrast to the time-consistent network topology (Section 4.2.3), the connectivity between actors in partial device participation changes throughout the learning procedure (i.e., in every epoch). Relying on the concept of information spread in gossip protocols, consensus between the actors can be achieved under basic assumptions [116], [117].

With partial device participation, we can reduce the communication in DFL by a constant factor. An exemplary scenario of three epochs of learning in a fully-connected network of 6 actors with random selection of 40% of neighbors is depicted in Figure 4.6. Without partial device participation, every actor would transmit 5 model parameter updates in each epoch, resulting in (6 actors $\cdot$ 5 transmissions) $=$ 30 transmissions per epoch. In contrast, with a participation ratio of 40%, this number reduces to (6 actors $\cdot$ (40% $\cdot$ 5 transmissions)) $=$ 12 transmissions.

Random partial device participation emerges as a simple but powerful strategy to reduce communication in DFL while maintaining reasonable convergence. Therefore, many DFL approaches adopt this random selection strategy in their system, employing a variety of participation rates [37], [39], [87]. Given the changed setting in approaches that leverage Mutual Knowledge Transfer for model aggregation, the authors in [29] apply random partial device participation to select both the senders and the receivers of model parameter updates in the system. Besides random selection of neighbors, more sophisticated strategies exist which try to further reduce communication overhead. To achieve this, these strategies focus on minimizing the participation ratio to communicate with fewer neighbors in each round while preserving convergence and selecting proper neighbors to increase convergence rate and thus reduce the number of epochs required to converge.

Collection	Commonality	Publications
I	Full device participation (Section 4.4.1)	[90], [91], [93], [94]
II	Dynamic number of local updates (Section 4.4.2)	[37], [39], [88]
III	Perfect parameter transmission (Section 4.4.3)	[29], [30], [38]
IV	Sparsification for compression (Section 4.4.4)	[87], [89], [92]

Table 4.6.: Grouping of publications into four distinct collections according to commonalities in implemented optimization techniques.

The authors in [89] and in [30] limit the communication to only a single neighbor per actor in each epoch. Moreover, in [89], they consider the respective inter-node bandwidth when selecting the neighbor but still allow some randomness in the selection process in order to facilitate convergence. Without this level of randomness, their strategy would always select the connection with the highest bandwidth, resulting in a permanent change of network topology in the case of consistent bandwidths. In [30], on the other hand, the authors select the most informative neighbor for the parameter transmission. More specifically, they pick the neighbor with the currently most divergent model parameters, measured by the cosine distance between the last layers of the model. By doing so, however, we require additional communication to retrieve the last layer of the model parameters from all neighbors. Therefore, the advantage of this approach regarding communication reduction highly depends on the size of the model and on the number of potential neighbors.

The loss-based strategy from Liao *et al.* [38] also requires additional communication, as the actors transmit their training loss (single value) to all connected actors directly after the local updating step. Based on these loss values and the communication frequencies to the individual connected actors, each actor then calculates priorities to select the set of activated neighbors in each epoch. Note that they embed a limit for the communication frequency in the priority calculations such that the priority of selecting a specific actor reaches its maximal value after a certain period without communication.

Unlike the selection of neighboring actors from the perspective of each individual actor (actor-centric selection), the authors in [88] and in [92] activate a subset of bi-directional links (network-centric selection). To achieve this, Wang *et al.* [88] decompose the base network topology into matchings (i.e., sets of disjoint links). They then assign an activation probability to each matching and optimize these probabilities to maximize the algebraic connectivity. Thereby, they prioritize connectivity-critical links, which eventually enhances the trade-off between convergence and communication. Similarly, in [92], the authors construct an active network topology on top of the consistent network topology in each epoch, trying to minimize the round time. To achieve this, they calculate the time required by each existing network link to remove slow network links iteratively from the base network topology. Simultaneously, they apply constraints based on the consensus distance to guarantee convergence, and adjust the individual compression ratios at the actors to comply with these constraints.

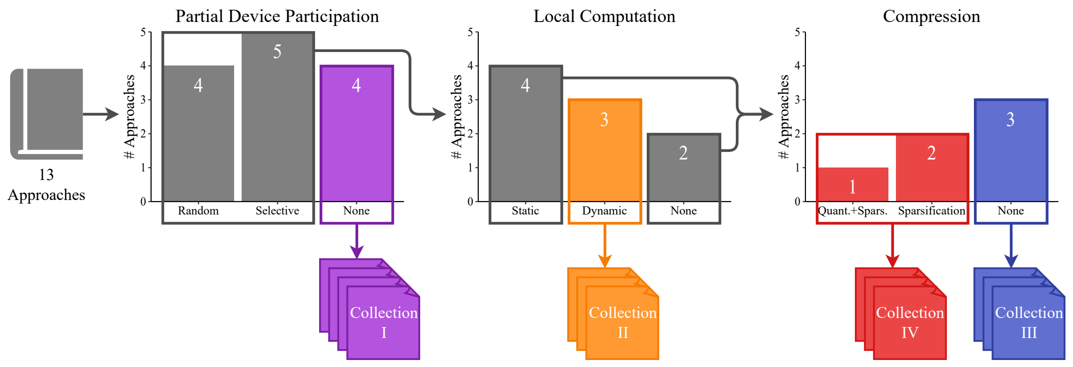


Figure 4.7.: Illustration of our extraction process for assigning the publications from the DFL literature to four coherent collections. The first collection extracts approaches that do not implement partial device participation; the second collection takes out the approaches that employ a dynamic strategy for local computation; and the third and fourth collections differentiate between whether or not they implement compression.

4.4. Communication Efficiency of DFL Approaches

In this section, we examine the interplay between the techniques outlined in the taxonomies from Section 4.2 and Section 4.3 with regard to communication efficiency by taking a holistic look at their implementations in common DFL approaches. In addition, we highlight the main objectives of the approaches and review their findings.

To avoid addressing each publication individually, we divide the 13 DFL publications discussed in this survey into four collections based on our taxonomy of communication optimization techniques. More specifically, we look at the distribution of the respective strategies to extract coherent sets from the 13 publications. The extraction process is illustrated in Figure 4.7. First, we consider the strategies from partial device participation, extracting the approaches that do not implement partial device participation (i.e., each actor involves all neighboring actors in the model exchange phase) to form Collection I. Next, we consider the strategies for local computation in approaches that implement partial device participation, and construct Collection II of approaches with dynamic numbers of local updates. Finally, the distribution of compression strategies naturally divides the remaining approaches into Collection III without compression and Collection IV in which the approaches apply sparsification. Table 4.6 lists the four collections including the publications they contain and their uniting commonality. By discussing the publications in these collections throughout the following subsections, we reveal typical combinations of the collection feature with other communication-influencing methods.

4.4.1. Collection I: Full Device Participation

Collection I comprises references [90], [91], [93], [94], which, in addition to implementing full device participation, all employ the synchronous Synchronization Method (Section 4.2.2). The combination of full device participation and a synchronous DFL system offers the advantage of relying on exactly one update from each neighboring actor in every communication round. In contrast, actors in asynchronous DFL must support varying numbers of received updates anyway, thus promoting the implementation of PDP. In [90], [91], [93], they follow the wAvg method for Model Aggregation (Section 4.2.1), whereas in [94], the authors follow the incAvg method. These four publications differ in their Network Topology (see Section 4.2.3), Compression method (see Section 4.3.1), and the number of local updates for Local Computation (see Section 4.3.2), covering a large portion of possible combinations with different strategies from these techniques. Wang *et al.* [90] perform 5 local updates, apply their novel sparsification method (TopK sparsification with momentum), and conduct experiments under several network topologies (fully-connected, random graph, and chain topology). In [91], the authors also implement a novel compression method, namely the Lloyd-Max quantizer with adaptive quantization levels. In their experiments, they perform 4 local updates, but we were unable to discover from the publication whether they use a random graph or a fully-connected network topology. In contrast to these two approaches, Gupta *et al.* [94] do not implement a compression method to reduce the size of the communicated model updates. Their primary innovation lies in selecting the best path through the network for the incAvg aggregation method, optimizing both accuracy and communication cost. They also implement Local Computation, considering the set of 1, 2, 5, and 10 local updates under the directed graph network topology. In [93], the authors do not facilitate any of the communication optimization techniques discussed in Section 4.3. However, they are optimizing the communication overhead by arranging the network topology into a torus topology. Within this framework, they employ a two-layer circular parameter synchronization strategy to reduce the overall frequency of parameter synchronization.

The experiments of all four publications in Collection I show a reduction in communication compared to their baselines. Wang *et al.* [90] evaluate the communication cost and the robustness to adversaries of their novel TopK sparsification method with momentum (MomTopK). They compare their results against FedAvg [1], standard TopK sparsification [107], and RandomK sparsification [118], evaluating the communication cost considering the convergence rate in terms of the Area Under the Curve and the F1-score metrics. Throughout their experiments, they set the sparsification ratio to 0.4 for all sparsification methods. This results in a communication reduction of 60% compared to FedAvg, with only a slight degradation in the convergence rate. Furthermore, they report that their approach converges the fastest among the baselines that implement sparsification. MomTopK converges almost twice as fast as the RandomK approach, whereas the TopK method seems to have difficulties converging in their experiments.

In contrast to measuring convergence speed in terms of the number of communication rounds in [90], the authors in [91] additionally consider the time progression under a fixed communication rate of 100 Mb/s, which is proportional to the number of bits communicated. In this way, they compare the communication cost of their quantization approach (LM-DFL) against the baselines of DFL without quantization, DFL with the ALQ quantizer [119], and DFL with the QSGD quantizer [120]. Similarly to the results in [90], DFL without compression shows the best convergence with the same number of iterations. Despite this, the convergence of LM-DFL outperforms the baselines in both the number of iterations and the time progression.

In [93] the authors evaluate the communication overhead based on the time cost required to perform a certain number of steps. Thereby, they compare their novel parameter synchronization strategy (Fedcs) against the gossip algorithm [116] under both the fully-connected and the torus network topology. They report the superiority of Fedcs over the baselines for scenarios with 16, 25, 64, and 100 devices in the network and a consistent number of $2(n-1)$ transmitted model updates per round. Additionally, they evaluate the reduction in communication overhead of their device placement algorithm in an ablation study under various network sizes, yielding optimization rates between 40% and 83% compared to a random placement.

Unlike all of the approaches above, in [94], they calculate the communication cost as the sum of edge weights over which a model update is transmitted, thus considering the bandwidth of the connection links. They demonstrate that their incremental training approach (TravellingFL) converges fast, hence, it lowers the communication bandwidth requirements as it needs by performing fewer communication rounds to converge. Compared to FedProx [121], Gupta *et al.* [94] demonstrate a decrease in communication cost of 23% to achieve similar accuracy in a network with 7 actors. In addition to comparisons with the baselines, they experimentally demonstrate the behavior of TravellingFL in numerous aspects in their publication, including the effect of local training epochs (i.e., Local Computation in Section 4.3.2), the scalability in terms of the number of actors, their selection of the optimal path through the network, and the performance of different model architectures.

4.4.2. Collection II: Dynamic Number of Local Updates

The publications in Collection II employ a dynamic number of local updates for Local Computation (see Section 4.3.2), either by randomly selecting this quantity from a prespecified range [39] or by indirectly allowing for consecutive local updates due to other techniques [37], [88]. In [88] and in [37], the authors employ a wAvg method for Model Aggregation (see Section 4.2.1, whereas in [39], they utilize the inexact alternating direction method of multipliers (iADM). These three publications cover all three types of Synchronization Methods (see Section 4.2.2), underlining the compatibility of dynamic Local Computation in synchronous [88], asynchronous [37], and semi-synchronous [39] DFL systems. The authors in both [39] and [37] implement a random selection of devices for Partial Device Participation (see Section 4.3.3), alongside the Compression technique (see Section 4.3.1) of compressed sensing or sparsification, respectively. In contrast, Wang *et al.* [88] do not involve Compression. Their novelty lies in their matching-based selection strategy (MATCHA) for Partial Device Participation, in which they decompose the network topology into a set of matchings[7], select a set of matchings that are activated, and optimize the selection to ensure fast convergence and minimize the communication delay. Activating only a subset of matchings opens up the possibility that some actors do not connect to a neighboring actor. In this case, the respective actors do not share their model update but continue to train for a consecutive local epoch, thus facilitating the technique of Local Computation implicitly. Similarly, Liu *et al.* [37] employ a random selection of one neighboring actor for transmitting the model update. Hence, actors that do not receive a model update from their neighbors perform another updating round without additional input, i.e., Local Computation. Note that even though the authors in [37] call the iterations of this updating round "local updates", we do not consider this as an implementation of Local Computation as it involves the aggregation of model updates from neighboring actors, thus not being strictly local. In [39], they

[7]A matching is a subgraph of the original network with a degree of one (i.e., each actor is connected to at most one neighboring actor).

draw a random number between 10 and 15 in each round to determine the number of local updates in one of their experiments, thus directly implementing a dynamic number of local updates.

The experiments of Wang *et al.* [88] focus on measuring the training loss over time and over the number of epochs. They compare various specifications of the communication budget for MATCHA against the baselines vanilla decentralized SGD [122], [123], periodic decentralized SGD [113], [123], and ChocoSGD [87]. The communication budget indicates the fraction of the reduction in communication per iteration compared to vanilla decentralized SGD. In their experiments, they find that a communication budget of 0.4 not only reduces the communication time by 2.5× but also results in a lower error than vanilla decentralized SGD. They also report a reduction in wall-clock time of 5× less than vanilla decentralized SGD with a communication budget of 0.02 to reach a certain target loss. Comparisons of MATCHA to the periodic decentralized SGD result in consistently achieving better convergence while preserving the same runtime per iteration. Additionally, they demonstrate that MATCHA can be used as a tool to improve other decentralized computation tasks, reporting a 2× speedup in wall-clock time when implementing MATCHA on top of ChocoSGD.

Zhou *et al.* [39] conduct an ablation study of their DFL approach (CEPS). By ablating their compression method and the integrated differential privacy mechanism (total of four combinations), they assess the behavior of CEPS under various number of nodes, values of the differential privacy parameter ε, and participation rates for Partial Device Participation. They report that enabling both 1BCS compression and differential privacy requires more iterations to converge when there are fewer actors. As the number of nodes increases, however, the differences diminish. Regarding the participation rate, their results show a minimal impact on the convergence of different values. A higher participation rate leads to an increase in computational time, which is compensated by a decrease in the amount of iterations needed to converge. In addition to this ablation study, Zhou *et al.* [39] compare CEPS with the baselines D-PSGD [122], DFedAvgM [124], and DFedSAM [125] in the second part of their experiments. By measuring the data transmission volume (DTV), they show that CEPS achieves a substantial reduction in DTV, saving at least 90% of the communication overhead compared to the baselines. Moreover, they demonstrate that increasing the participation rate leads to fewer iterations but higher DTV. CEPS requires significantly fewer communication rounds to converge than the baselines, therefore consistently achieving the highest communication efficiency.

Liu *et al.* [37] compare their DFL approach (AEDFL) against 14 baselines, namely FedAvg [1], FedProx [126], FedNova [75], SAFA [127], Sageflow [128], AD-PSGD [129], FedSA [130], ASO-Fed [131], FedBuff [132], Port [133], HRank [134], FedAP [135], HAP [136], and DisPFL [35]. The experiments consider a network of 100 actors which are connected in an exponential graph network topology, and the data was partitioned using the Dirichlet distribution [68]. AEDFL achieves the highest accuracy and significantly outperforms the baselines in terms of training speed. Their experiments comprise several image classification tasks as well as one natural language processing task. In all of these tasks, AEDFL shows a reduction in training time between 52.2% and 92.8%, and a reduction in computation cost between 6.9% and 42.3% compared to the baselines. Unfortunately, their experiments do not explicitly evaluate the impact of their sparse model training technique on the communication cost.

4.4.3. Collection III: Perfect Parameter Transmission

Collection III comprises the subset of three approaches from the unassigned publications that do not compress the parameters in the model update before sending them over the network, thus employing perfect parameter transmission [29], [30], [38]. Instead of optimizing communication cost through Compression, all three approaches implement Partial Device Participation (see Section 4.3.3) and static Local Computation (see Section 4.3.2) to enhance communication efficiency. Li *et al.* [29] select 20% of the actors to participate in training and conduct experiments with 1 and 10 local updates at the actors. In [38], the authors introduce a loss-based strategy for Partial Device Participation and perform two consecutive local updates in their experiments. Soltani *et al.* [30] also propose a novel technique for Partial Device Participation. In contrast to [38], each actor selects only a single neighboring actor to participate in the parameter exchange based on the last layer model similarity. Their experiments involve a fixed quantity of five local updates per communication round. All three approaches of Collection III differ in their network topology. In [29], the authors consider the popular fully-connected topology. In [38], on the other hand, they employ a random, directed network topology with a minimum amount of incoming neighbors for each device, and in [30], the authors perform their experiments using both the grid topology and the hybrid topology called Ring of Cliques. When it comes to the Model Aggregation method (see Section 4.2.1), the authors in [38] and in [30] implement the popular type of wAvg. Conversely, Li *et al.* [29] propose model fusion by transferring learned knowledge between devices to address the problem of data heterogeneity. Similarly outstanding is the Synchronization Method (see Section 4.2.2) by Liao *et al.* [38], as they propose an asynchronous DFL system while the authors in [29] and in [30] implement the synchronous protocol.

In [29], the authors compare their approach (Def-KT) to the baselines FullAvg (as implemented in [137]–[139]) and Combo [116] by conducting experiments on four different datasets (MNIST [140], Fashion-MNIST [141], CIFAR-10 [142], and CIFAR-100 [142]) with two model types (multilayer perceptron and convolutional neural network) under both homogeneously partitioned datasets and heterogeneously partitioned datasets with various degrees of heterogeneity. To create a proper foundation for comparison, they configure all approaches to have the same communication overhead per round. That way, they perform several experiments to evaluate the impact of different numbers of actors in the network, the effect of various amounts of local updates, and the differences in training caused by heterogeneous partitioning. The results show that, in most cases, Def-KT converges faster and attains a higher accuracy than the baselines. Additionally, Def-KT is more reliable than the baselines, since the accuracy oscillates less severely under the heterogeneous scenario. The evaluation of local accuracy reveals that Def-KT significantly outperforms the baselines, and the gap in local accuracy between Def-KT and the baselines enlarges with an increasing degree of heterogeneity. Hence, the proposed method is more robust to heterogeneous data.

The authors in [38] also perform experiments with both homogeneously and heterogeneously partitioned datasets. They compare their asynchronous DFL approach (AsyDFL) to the baselines AsyNG [28] (AsyDFL is an extension of AsyNG), AGP [143], OSGP [144], and NetMax [145], performing image classification tasks with two different models (AlexNet [146] and VGG16 [147]) on two datasets (EMNIST [148] and ImageNet-100, a subset of ImageNet [149]). Their performance metrics include test accuracy, training loss, network traffic, and the time to converge. In their evaluations, AsyDFL achieves the highest accuracy compared to the baselines in both the homogeneous and heterogeneous data scenario after a fixed time. Furthermore, the experiments indicate that the superiority of AsyDFL over the baselines increases with higher data heterogeneity. In terms of communication overhead, they demonstrate a reduction by 54% and 71% in network traffic consumption to achieve a certain accuracy, compared to the baselines AGP and OSGP, respectively. Comparisons of the completion times to reach a target accuracy reveal the superiority of asynchronous approaches in general, as the synchronous baseline OSGP requires almost twice as much time to reach the target accuracy than AsyDFL and the other baselines. In summary, Liao *et al.* [38] conclude their experiments with a reduction in communication cost by nearly 57% and a decrease in completion time by about 35% compared to the baselines.

Similar to the approaches in [29] and [38], Soltani *et al.* [30] conduct experiments under different degrees of data heterogeneity. They compare their DFL approach (DFLStar) to the baselines D-PSGD [122], PASGD [123], and DFLStar-RS (a variant of DFLStar with random Partial Device Participation instead of their novel selection strategy) by performing image classification with a convolutional neural network on two datasets: CIFAR-10 and CIFAR-100 [142]. To assess the effectiveness of DFLStar, they consider test accuracy, training time, and communication traffic in their experiments. Evaluations of the test accuracy over time demonstrate that DFLStar consistently outperforms the other baselines, maintaining the best performance across all degrees of data heterogeneity for both network topologies. Comparisons of the communication cost to achieve a target accuracy indicate a cost reduction of approximately 96% for D-PSGD and approximately 81% for PASGD. Therefore, DFLStar significantly minimizes the communication cost. As a natural implication, they also report a decrease in training time compared to D-PSGD and PASGD of up to 78% and up to 55.8%, respectively. Soltani *et al.* [30] additionally conduct an ablation study to assess the benefit of their integrated self-knowledge distillation and their selection strategy for Partial Device Participation, resulting in an increase in accuracy of 5.2% from the self-knowledge distillation and up to 2.45% from the selection strategy. This successfully demonstrates the advantages of DFLStar and the implemented techniques.

4.4.4. Collection IV: Sparsification for Model Compression

The remaining three common approaches constitute Collection IV, finding their commonality in employing sparsification for Compression (see Section 4.3.1) before sending the model update over the network [87], [89], [92]. Since these three approaches focus on advancing communication optimization techniques, they maintain simplicity with regard to other DFL components by implementing the popular wAvg strategy for Model Aggregation (see Section 4.2.1) with the synchronous Synchronization Method (see Section 4.2.2). In addition to experimenting with a fully-connected Network Topology (see Section 4.2.3) in all three publications, Koloskova *et al.* [87] consider the ring topology and the torus topology, and Wang *et al.* [92] consider the ring topology. Only one of the publications in Collection IV, namely the approach in [92], utilizes Local Computation (see Section 4.3.2) with 50 local updates per round. The key contributions by Koloskova *et al.* [87] are a novel gossip-based stochastic gradient descent algorithm and a novel gossip algorithm with support for arbitrary compression under linear convergence. They consider the technique of Partial Device Participation (see Section 4.3.3) in the form of an exemplary compression operator that randomly removes model updates with a given probability. However, they do not include this operator in their experiments. In [89], the authors propose a novel gossip matrix generation algorithm to optimize bandwidth utilization, thus introducing a new technique for Partial Device Participation. Here, each actor communicates with only one neighboring actor, transmitting a model update that is compressed using a global sparsification mask. Employing the same communication optimization categories, Wang *et al.* [92] tackle the challenge of slow convergence rate due to fixed communication topologies and static compression ratios. More specifically, they propose an algorithm to optimize the selection for Partial Device Participation and to determine the compression ratios dynamically for each actor.

The experiments in [87] are divided into two parts: The first part evaluates their gossip approach (CHOCO-G) and the second part experiments with their stochastic gradient descent technique (CHOCO-SGD). Both experiment sections rely on the datasets epsilon [150] and rcv1 [151], homogeneously and heterogeneously partitioned among the actors. The baselines to assess the performance of CHOCO-G include the gossip protocol with exact communication [117], PQDA [152] (named Q1-G in their experiments), and the scheme Q2-G, which was proposed by Carli *et al.* [153]. The corresponding experiments focus on the number of iterations and the number of transmitted bits in contrast to the training error. In this setting, Koloskova *et al.* [87] conduct experiments with random quantization QSGD [120], RandomK sparsification [108], and TopK sparsification [108]. Their experiments show that, with the QSGD quantizer, CHOCO-G converges at the same rate as exact gossip while significantly outperforming Q1-G and Q2-G. With RandomK sparsification, CHOCO-G also proves better than Q1-G and Q2-G, yet 100× slower than exact gossip due to the 1% sparsification ratio. In terms of transmitted bits, however, CHOCO-G shows the same convergence speed as exact gossip. Additional experimental results demonstrate that using TopK sparsification can further enhance convergence. To evaluate the performance of CHOCO-SGD, they compare to using exact communication, and to the baselines DCD-PSGD and ECD-PSGD from [154]. They report that CHOCO-SGD achieves almost the same convergence rate as the exact algorithm with a reduction in communication of 100× when using Random-1% sparsification and 15× when employing QSGD with 4-bit precision. CHOCO-SGD consistently outperforms DCD-PSGD and ECD-PSGD in the experiments.

The experiments of [89] and [92] both include CHOCO-SGD [87] as one of their baselines. Tang *et al.* [89] conduct experiments in two settings: real-world bandwidths with 14 actors and random bandwidths with 32 or 100 actors. They consider image classification tasks using convolutional neural networks including ResNet-20 [155] on the datasets MNIST [140] and CIFAR-10 [142], homogeneously and heterogeneously partitioned among the actors. They report the convergence performance, communication efficiency, and bandwidth utilization of their approach (GossipFL), comparing to the baselines FedAvg [78], S-FedAvg [1], D-PSGD [122], DCD-PSGD [154], and CHOCO-SGD [87]. The experiments of reference [89] reveal that GossipFL achieves comparable convergence with D-PSGD in the homogeneously partitioned data setting, yet outperforms all baselines with heterogeneously partitioned data. They also show that both GossipFL and CHOCO-SGD preserve the convergence performance on homogeneously partitioned data at high compression ratios. In terms of communication cost, their findings demonstrate a significant reduction in communication traffic while preserving the model accuracy. For training the ResNet-20 model on CIFAR-10, they report 52% less communication traffic than FedAvg and 34% less than S-FedAvg to reach the same accuracy. For the MNIST dataset with its respective model, they report a reduction in communication traffic of 40% and a decrease in required communication time of 19% compared to S-FedAvg. Further large-scale experiments with 32 and 100 actors on heterogeneously partitioned data record a saving in communication cost of 14× compared to FedAvg, 16× compared to CHOCO-SGD, and 154× compared to DCD-PSGD. The end-to-end saving in communication time is approximately 10× compared to state-of-the-art solutions.

Wang *et al.* [92] experiment with the models VGG9 [156] and ResNet-9 on the datasets CIFAR-10 and CIFAR-100 [142], respectively. Their setup consists of 20 actors that are connected with different, fluctuating bandwidths to simulate LAN and WAN connections. They evaluate the test accuracy, training time, and traffic consumption of their approach (CoCo) compared to the baselines DCD-PSGD [154], CHOCO-SGD [87], and SASP [118]. Additionally, they compare the performance to logically centralized training (LCT) where all links are activated without model compression, serving as an upper bound for the accuracy of DFL in general. In their experiments, CoCo shows a speedup in training by 10× and a reduction in communication traffic of 50% on average for homogeneously partitioned data compared to the baselines. CHOCO-SGD emerges as the most communication-efficient approach among all approaches, yet converging about 3× slower to a lower accuracy than CoCo. For different levels of heterogeneity in the data partitioning, CoCo attains an acceleration in training from 6.5× to 13.8× and a reduction in traffic consumption of 40% up to 68% compared to the baselines. Wang *et al.* [92] also report that CoCo reaches similar accuracy as LCT. Moreover, with heterogeneously partitioned data, the accuracy of CoCo decreases only by 1.7% while the accuracies of the baselines decrease by $3.7\% - 9\%$, implying that CoCo is more robust to heterogenous partitioning. Further experiments in a simulated real-world environment on the SVHN dataset [157] record a speedup from 2.9× up to 3.8× to achieve the same accuracy with a reduction in traffic consumption of $63\% - 67\%$. Similar experiments on the STL10 [158] dataset result in a speedup between 1.2× and 2.1× with a reduction in communication traffic of $11\% - 44\%$. These experiments further demonstrate the effectiveness and efficiency of CoCo in real-world scenarios.

4.5. Discussion

Based on the established taxonomy of key DFL components critical to communication efficiency, we analyzed the impact on communication efficiency within the general framework of primary strategies (see Section 4.2). It becomes apparent that the respective network topology exerts the most direct effect on communication cost by determining the connectivity between actors. In contrast, individual model aggregation strategies and synchronization methods show a strong indirect influence on communication cost through the convergence rate. We further categorized popular techniques to optimize communication in DFL and examined their impact on communication efficiency in Section 4.3. Our findings highlight the versatility of compression and local computation techniques, given their orthogonality to other DFL constituents and their excellent tuning capability. To identify methodological combinations addressed in existing research, Section 4.4

explored the interplay of strategies from our taxonomies in concrete realizations. In addition to providing a detailed review, this analysis highlights underexplored research directions. For instance, integrating compression methods with knowledge transfer for model aggregation and full device participation presents a promising combination for future work. Furthermore, methods for dynamically determining the number of local updates for local computation are currently under-researched, suggesting an avenue for developing sophisticated strategies to enhance communication efficiency in DFL.

Our summary of the experimental findings of DFL approaches elucidates the overarching impact of their implemented strategies on both communication efficiency and performance. However, there is no universal approach to reducing communication cost in a DFL system, given the dependence on the overall system configuration, such as data properties, model properties, and system components, as well as the numerous trade-offs between the challenge of communication efficiency and other challenges. To clarify these trade-offs, we discuss the effect of strategies to address the core challenges in DFL (as listed in Section 2.3) on communication efficiency in the following paragraphs.

Security and privacy × communication efficiency In DFL, additional measures are often integrated to address security and privacy risks that the distributed and decentralized design alone cannot mitigate (see Section 2.3). Those methods can influence communication efficiency in various ways. Fault-tolerant DFL systems that are robust to node failures or dropouts often require more connections to provide redundancy. Cryptographic methods for node verification or blockchain-based systems require the exchange of additional information such as cryptographic proofs or transaction metadata. Depending on the method, detecting poisoning attacks may also necessitate additional communication, although some approaches eliminate this need [19]. Integrating privacy-enhancing technologies into DFL systems can also significantly influence communication efficiency, as the pursuit of stronger privacy guarantees usually involves increased communication overhead or a reduction in overall system performance. Under multi-party computation (MPC), each node splits its update into secret shares and distributes them among a set of peers, which substantially increases communication overhead as multiple messages have to be transmitted. However, more communication-efficient variants have been proposed [21]. Differential privacy (DP) is most commonly applied locally: Each node adds carefully calibrated noise to its model updates before sharing them with peers, thus ensuring that individual data points cannot be inferred from the updates. While this does not change the size of the model updates, it can lead to slower convergence which, in turn, results in a higher number of training rounds to achieve acceptable accuracy. This effect can be reduced by using notions of DP that leverage the properties of DFL to amplify privacy guarantees [24], [25]. When using homomorphic encryption (HE), each node typically encrypts its local model updates and aggregation is performed on the encrypted data. Encryption increases the size of model updates, thereby increasing the communication cost. Furthermore, decentralized key management, e.g., multi-key HE [26], requires the exchange of additional information, further adding to communication overhead.

Data heterogeneity × communication efficiency Heterogeneous data distributions between actors can cause local model parameters to converge at different stationary points, thereby hindering global convergence [79]. A straight-forward approach to address this concern is to communicate in a dense network (e.g., construct a fully-connected network topology with full device participation) and to increase the frequency of synchronization among the actors (i.e., lower the number of local updates). This, however, conflicts with the challenge of communication efficiency, causing communication cost to spiral up drastically. Therefore, there exists a general trade-off between communication efficiency and the challenge of data heterogeneity. Work to optimize this trade-off includes the development of specific aggregation methods [31], [32], [38], the design of heterogeneity-aware network topologies [159], and the consideration of data heterogeneity for partial device participation [28], [38], [92]. However, some of these approaches impose tight assumptions on the DFL scenario (e.g., assuming knowledge about the distributions of labels at other actors [159]), or even contradict the DFL paradigm by exchanging data (e.g., sharing synthetic data [32]). Others exchange supplementary information among the actors to prevent their gradients from drifting apart (e.g., sharing the training loss [28], [38] or estimated consensus distances [92] to aid the partial device participation strategy). Such supplementary information is typically in an aggregated state and significantly smaller than a gradient or a full set of model parameters. Hence, exposing aggregated statistics about the current state of the actor to detect gradient drifts emerges as a promising direction to avoid frequent synchronizations when addressing data heterogeneity. The main challenge here is to correctly detect gradient drifts based on the aggregated statistics. This detection method could then be applied to methods such as local computation (see Section 4.3.2), partial device participation (see Section 4.3.3), or other DFL constituents to dynamically prevent actors from diverging.

System heterogeneity $\times$ communication efficiency The most common approach to overcome the problem of stragglers, arising from differences in computing and communication resources among actors, is to enable asynchronous [37], [38] or semi-synchronous [39] training. However, this leads to an increase in communication overhead compared to synchronous DFL, as discussed in Section 4.2.2. Moreover, asynchronous and semi-synchronous DFL systems need to account for stale model updates. A standard approach to mitigating the effects of staleness is to either exclude outdated model updates from the aggregation or to incorporate the updates with reduced weights, for example by assigning an advancement-dependent weighting factor in the aggregation rule [37]. This, in turn, leads to considerable communication overhead, as these model updates have been transmitted but are then not used to their full extent. Consequently, a fundamental trade-off arises between the challenges of communication efficiency and the challenge of system heterogeneity. The more we optimize a DFL application for communication efficiency, the more leeway we give to the problem of stragglers—and thus potentially undermine a slow actor. Conversely, if the DFL system focuses on overcoming the problems of system heterogeneity, the transmitted information is not fully exploited, thereby introducing redundant communication overhead.

5. Dynamic Synchronization Rule: Gradient Thresholding

5.1. Introduction

In FL, each device periodically computes an update to its local model parameters, which is then transmitted over the network and incorporated into the model parameters of other nodes. This exchange of model updates forms the main source of communication cost. Synchronous DFL and semi-synchronous DFL (see Section 4.2.2) comprise strict synchronization points that trigger the exchange of model updates on all actors. Therefore, each synchronization point generates an associated increase in communication cost.

Reducing the frequency of model update exchanges is an effective technique to address the challenge of communication efficiency in DFL. However, periodic sharing of model updates is crucial to achieve convergence and foster collaboration between devices. It is therefore important to choose synchronization points wisely in order to maintain convergence while keeping communication to a minimum. Additionally, the heterogeneous nature of the datasets of individual actors in real-world deployments demands great responsiveness to suddenly divergent actors. This raises the challenge of minimizing the number of synchronization points while maintaining convergence, enabling collaboration, and reacting to divergent actors.

In this chapter, we propose Gradient Thresholding—a novel synchronization rule to tackle this challenge in the setting of synchronous DFL with a fully-connected network topology. Motivated by the importance of finding a universally applicable method to address this challenge, we investigated the evolution of gradients (i.e., updates to the model parameters) during training [160]. The observation that—with small enough learning rate—gradients follow a certain direction in the parameter space ignites the idea of detecting divergent actors based on a deviation from their previous path in the parameter space. Therefore, we construct a region in the parameter space to delimit the area in which actors are allowed to enhance their model parameters without requiring synchronization with other actors. We call this area the threshold region. The position and extent of the threshold region depend on the previously synchronized gradients and are therefore the same speculative measure for all actors. As long as the model parameters of the individual actors remain within this threshold region, we presume that the actors follow a common training path, and thus do not need to synchronize with each other. As soon as an actor surpasses the threshold region, however, it might have diverged from the other actors and, thus, triggers synchronization between the actors. Thereby, Gradient Thresholding eliminates redundant synchronization points while reacting to divergent actors. Our experiments demonstrate that Gradient Thresholding outperforms four baseline in several cases.

5.2. Background, Challenges, and Related Work

5.2.1. Challenges

Communication Cost Minimization Synchronizing the model parameters among actors requires exchanging the respective model updates over the network, which incurs communication cost. Therefore, the number of synchronizations must be reduced to minimize communication cost. However, synchronization of model parameters prevents actors from striving for different optima (i.e., actor drift) and, thus, ensures convergence. Insufficient synchronization during the DFL process jeopardizes convergence, while excessive synchronization causes the communication cost to skyrocket. This trade-off raises the open research challenge to find the optimal synchronization points.

Divergence Detection A straight-forward approach to the challenge of minimizing communication cost is to synchronize the model parameters as soon as the actors start to diverge from each other in their local model parameters. Divergent model parameters are caused by actors optimizing toward different solutions due to their heterogeneous data distributions. This scenario is also known as actor drift in DFL, client drift in CFL, or concept drift. Whether different sets of model parameters diverge from each other or strive for same optimum on displaced paths depends on the underlying loss surface. Since the shape of this loss surface is unknown, it is hard to distinguish between the two cases. Therefore, the challenge arises to detect actor drift.

Limited Communication In addition to the challenge of detecting and preventing actor drift, there is the restriction in communication. Since our objective is to reduce the communication cost in DFL, constantly comparing the actors' model parameters to check whether they diverge would induce unfavorable communication. Thus, the actor is limited to the information that is communicated during the previous synchronization. This complicates the detection of divergent model parameters considerably, as the actors are unaware of the training progress made by their neighboring actors.

5.2.2. Related Work

Local computation (sometimes also referred to as local SGD or local-update SGD) is a popular strategy to reduce the number of communication rounds in DFL. With local computation, every actor performs multiple consecutive updates in-between synchronization points instead of synchronizing after every local update [1], [112]. This strategy thereby aims to optimize the trade-off between communication efficiency and performance. We further detail the strategy of local computation in Section 4.3.2.

To skip communication rounds and, thus, reduce the number of synchronization points, Chen *et al.* [161] proposed to reuse the lagged gradient if the new gradient did not change much (slowly-varying). They present two different approaches for detecting qualifying gradients: "LAG-WK" and "LAG-PS". Despite the fact that this approach depends on a central server, both approaches conflict with our challenge of limited communication. In "LAG-WK", the server needs to broadcast the current model parameters to the nodes in every epoch, so that the nodes can decide whether their gradient should be updated on the other nodes. In "LAG-PS", on the other hand, the central server obtains the estimated smoothness constant of the local objective functions from all nodes in every round, implying additional communication.

Zhang *et al.* [162] proposed the Adaptive Synchronous Parallel Strategy, which also involves a central server, but could be adapted to DFL with a lightweight coordinator. In their strategy, a synchronization barrier, controlled by the central server, prevents the nodes from synchronizing in every round. The server constantly monitors the system performance and iteration counts of the nodes to open the synchronization barriers only if a node exceeds certain thresholds based on the staleness and weakness of its model. However, consistent performance monitoring requires the transmission of the corresponding information, which conflicts with the challenge of limited communication. Additionally, this strategy is not designed to address client drift caused by heterogeneous datasets among the nodes.

Kamp *et al.* [163] proposed a protocol for CFL to reduce the number of synchronizations while quickly adapting to concept drifts. To achieve this, each node monitors the euclidean distance between its local model parameters and a reference model that is common among all nodes (i.e., the model parameters after the most recent synchronization). As soon as this distance exceeds a given threshold value, the coordinator performs a partial synchronization—trying to find an averaged model that falls within the threshold—or a full synchronization, which also updates the reference model. However, this approach is based on the distance to the reference model and, hence, triggers synchronization even if the individual model parameters are similar but evolving away from the reference model.

Theologitis *et al.* [164] approaches this problem by considering the variance of the local model parameters in CFL. To estimate this variance, however, information about all local model parameters is needed. This, in turn, requires additional communication from the clients to the server. Although they provide solutions for estimating the variance with minimal communication needed, adding additional communication still conflicts with our challenge of limited communication.

Our approach—Gradient Thresholding—addresses all three of the above challenges simultaneously. To our knowledge, this is the first approach for DFL that tackles all of these challenges. Yet, there are overlaps with related work. Similarly to Zhang *et al.* [162] and Kamp *et al.* [163], Gradient Thresholding uses a threshold that triggers synchronization of all participants when exceeded. The main difference in this regard is that Gradient Thresholding monitors compliance with the threshold on each device individually, whereas the other approaches do so centrally, taking into account multiple devices. As in Kamp *et al.* [163], the threshold in Gradient Thresholding is based on the distance to the model parameters.

5.3. Methodology

Gradient Thresholding aims to minimize the number of synchronization steps while ensuring convergence of the participating devices. To achieve this, we only perform synchronization when apparently necessary, adapting to emerging disparities between the devices (i.e., actor drift). However, the fact that devices are isolated from each other except during synchronization poses the challenge of detecting disparities without communication between devices. To overcome this challenge, our approach exploits the information exchanged during synchronization to construct a threshold region in the parameter space. This threshold region defines the zone in which the devices are allowed to train their models independently (i.e., without synchronization). As a result, each device performs multiple model updates until their model parameters exceed the threshold boundary. When the threshold region is exceeded on a particular device, the device calls for synchronization. This triggers the synchronization of model parameters across all devices, resulting in synchronized model parameters and a new threshold region, which are the same for each device. Thereafter, the devices continue their independent training until a violation of the new threshold triggers the next synchronization. This process repeats until a certain stopping criterion (e.g., reaching the maximum number of epochs) is met. By applying Gradient Thresholding, we can react to divergent actors without the need for constant information exchange. Consequently, we omit synchronizations as long as the devices seek the same optimum, eliminating the communication cost of exchanging model parameters for redundant synchronization.

5.3.1. Threshold Region

The threshold region defines the zone in the parameter space in which actors can update their model parameters without requiring synchronization. We construct the threshold region based on the forecast of the next synchronized gradient, the number of epochs since the last synchronization, and the hyperparameter to control the extent, the decay, and the direction momentum. Since the threshold region spans only a small portion of the whole parameter space, proper design of the threshold region is crucial to perform multiple consecutive updates autonomously on the devices. If we specify the threshold region too tight or in the wrong direction, the devices quickly exceed its boundary, leading to frequent and unnecessary synchronization steps. Conversely, if we specify the threshold region too broad, synchronization is performed rarely and, thus, the model parameters at the devices could diverge drastically. The following paragraphs elaborate on the key aspects that characterize the threshold region.

Threshold Center The center of the threshold region defines the direction in which the threshold region extends, and influences the overall size of the region. In other words, the threshold center represents the position in the parameter space around which the devices can train independently without synchronizing with each other. Hence, this center reflects the expected parameter state for the next parameter synchronization. To construct the threshold center, we forecast the direction of the upcoming synchronized gradient by applying an exponentially weighted moving average on the lagged directions of the synchronized gradients, as shown in Equation 5.1. Here, the accumulated gradient since last synchronization for actor i is denoted as $\mathcal{G}_i^{\Sigma}$, and the k-th synchronized (i.e., averaged) gradient of all actors is denoted as $\overline{\mathcal{G}_{(k)}^{\Sigma}}$ accordingly. We compute the k-th gradient forecast $\widetilde{\mathcal{G}}_{(k)}$ as the sum of the current synchronized gradient direction $\overline{\mathcal{G}_{(k)}^{\Sigma}} \;/\; \left\|\overline{\mathcal{G}_{(k)}^{\Sigma}}\right\|_2$ weighted by θ_β and the direction of the $(k-1)$-th gradient forecast $\widetilde{\mathcal{G}}_{(k-1)} \;/\; \left\|\widetilde{\mathcal{G}}_{(k-1)}\right\|_2$ weighted by $(1-\theta_\beta)$. The coefficient θ_β, with a value between 0 and 1, represents the degree of weighting decrease for the exponential moving average. The resulting vector is scaled to the same length as the current synchronized gradient $\overline{\mathcal{G}_{(k)}^{\Sigma}}$

using the euclidean norm $\|\cdot\|_2$. Through these calculations, we obtain a forecast for the direction of the next synchronized gradient. This forecast forms the center of our threshold region.

$$\widetilde{\mathcal{G}}_{(k)} = \begin{cases} \overline{\mathcal{G}^{\Sigma}_{(1)}} & k = 1 \\ \dfrac{\left\|\overline{\mathcal{G}^{\Sigma}_{(k)}}\right\|_2}{\left\|\theta_\beta \frac{\overline{\mathcal{G}^{\Sigma}_{(k)}}}{\left\|\overline{\mathcal{G}^{\Sigma}_{(k)}}\right\|_2} + (1-\theta_\beta)\frac{\widetilde{\mathcal{G}}_{(k-1)}}{\left\|\widetilde{\mathcal{G}}_{(k-1)}\right\|_2}\right\|_2} \left(\theta_\beta \frac{\overline{\mathcal{G}^{\Sigma}_{(k)}}}{\left\|\overline{\mathcal{G}^{\Sigma}_{(k)}}\right\|_2} + (1-\theta_\beta)\frac{\widetilde{\mathcal{G}}_{(k-1)}}{\left\|\widetilde{\mathcal{G}}_{(k-1)}\right\|_2}\right) & k > 1 \end{cases} \tag{5.1}$$

Threshold Extent Building on the center of our threshold region, we define the region itself. The distance between the current synchronized gradient and the threshold center is exactly the length of the synchronized gradient. Therefore, the threshold region should span at least the length of the synchronized gradient in the corresponding direction to include the current state. Our threshold region is composed of two components: the distance along the line spanned by vector of the gradient forecast $\widetilde{\mathcal{G}}$ (the projected distance) and the distance orthogonal to it (the projection distance). Considering these two components, we construct a cylindrical threshold region with its main axis in the direction of the gradient forecast. We set the threshold for the projected distance to the length of the synchronized gradient. Thus, the threshold region extends from the synchronized gradient to the threshold center and the equivalent length beyond the threshold center. To avoid entering a state of permanent synchronizations, we additionally apply the factor $\rho = 1 + \frac{1}{k-k_s}$ to this threshold, where k is the current epoch and k_s is the epoch of the previous synchronization. Thereby, we allow the threshold for the projected distance to increase by up to twice the size, depending on the frequency of synchronizations. The threshold for the projection distance controls the width of the cylindrical threshold region. To account for different magnitudes in the model parameters when applying this threshold, we leverage a weighted euclidean distance. The weights W of this distance are determined based on the gradient forecast, essentially penalizing dimensions that show little change stronger than dimensions that show significant change. This weighted distance is further detailed in a separate paragraph below. Given that the weighted distance penalizes the dimensions according to the change in the gradient forecast, we set both the projected threshold and the projection threshold to depend on the length of the gradient forecast. On top of that, we introduce the hyperparameter θ_ρ that allows further scaling of the projection distance threshold. Adjusting this hyperparameter is analogous to setting the sensitivity of the threshold region. Given a specific gradient $\mathcal{G}$ in the parameter gradient space, the two thresholds are defined in Equation 5.2. Here, $\operatorname{proj}_{[\widetilde{\mathcal{G}}]}(\mathcal{G}) = \frac{\mathcal{G}\cdot\widetilde{\mathcal{G}}}{\widetilde{\mathcal{G}}\cdot\widetilde{\mathcal{G}}}\widetilde{\mathcal{G}}$ denotes the orthogonal projection of gradient $\mathcal{G}$ onto the line spanned by the gradient forecast $\widetilde{\mathcal{G}}$, and $\|\cdot\|_W$ denotes the weighted euclidean distance with weights W.

$$\begin{aligned} \left\|\operatorname{proj}_{[\widetilde{\mathcal{G}}]}(\mathcal{G}) - \widetilde{\mathcal{G}}\right\|_2 &\leq \left(1 + \frac{1}{k-k_s}\right)\left\|\widetilde{\mathcal{G}}\right\|_2 \\ \left\|\mathcal{G} - \operatorname{proj}_{[\widetilde{\mathcal{G}}]}(\mathcal{G})\right\|_W &\leq \theta_\rho\left(1 + \frac{1}{k-k_s}\right)\left\|\widetilde{\mathcal{G}}\right\|_W \end{aligned} \tag{5.2}$$

The resulting threshold region is visualized in Section 5.4.4 by means of a simplified experiment.

Threshold Decay To prevent the actors from diverging to different local optimums which lie inside the threshold region, we decrease the threshold width in every epoch. This resolves two concerns in Gradient Thresholding. First, the exponential decay ensures that synchronization will definitely occur again at some point in time. Thus, it addresses the case of actors that slow down drastically during training due to flatter areas in the loss landscape. Second, the synchronized gradient point disappears from the threshold region after a few rounds of training. While this does not sound particularly advantageous at first, it responds to the case where actors find a local optimum around the synchronized gradient and helps to re-adjust the threshold extent when needed. Gradient Thresholding applies the threshold decay to the threshold region in the form of a factor multiplied to both threshold components (i.e., the projected distance threshold and the projection distance threshold). The decay factor is specified by means of the hyperparameter θ_α and lies in the range $\theta_\alpha \in (0, 1]$.

Weighted Distance The elements of the model parameters show different sensitivities and magnitudes with respect to the performance of the model. Thus, making a small change on a sensitive parameter element could have a significant impact on model performance, whereas making the same change to an insensitive parameter element would have little effect. Therefore, distances in the parameter space must take into account the sensitivity of parameter elements. The simplest solution to overcome this issue would be to scale the parameter space according to the individual sensitivities of its dimensions. However, these sensitivities are unknown. We therefore calculate distances in the parameter space using a weighted euclidean distance. The weights are determined on the basis of the forecast gradient, which serves as a reference for the expected parameter sensitivities. Parameter elements that experience little change in the forecast are assumed to have higher sensitivity than elements that experience substantial change. Thus, we penalize sensitive parameters more than insensitive parameters by specifying the weights as the element-wise inverse of the absolute change in the forecast gradient. However, since certain model parameters might also remain constant in our forecast, the weights pose the problem of over-penalizing sensitive elements. Given the observation that elements in the same layer of a neural network are interdependent, we therefore limit the weights to a maximum of their layer-wise median. Equation 5.3 shows the complete calculation of the weights W for our weighted euclidean distance based on the forecast gradient $\widetilde{\mathcal{G}}$, with $A^{\circ -1}$ being the Hadamard inverse of matrix A, $A^{|\cdot|}$ being the element-wise transformation to absolute values, $\max_\circ(A, s)$ being the element-wise maximum of matrix A and scalar s, and $\operatorname{med}(A)$ being the median of the elements in matrix A.

$$W = \max_{\circ}\left(\widetilde{\mathcal{G}}^{|\cdot|}, \operatorname{med}\left(\widetilde{\mathcal{G}}^{|\cdot|}\right)\right)^{\circ -1} \tag{5.3}$$

5.3.2. Algorithm

Using the threshold region defined in Section 5.3.1, we define the synchronization rule of Gradient Thresholding. In this section, we provide the pseudo-code of the complete system and discuss in which aspects it differs from common DFL systems.

Learning Protocol Algorithm 1 provides the main protocol of our DFL learning process. The first loop on Algorithm 1 line 2 initialized the variables at the actors. In addition to the usual initialization of model parameters, we initialize the required state variables, including the accumulated gradient $\mathcal{G}_i^{\Sigma}$, the forecast gradient $\widetilde{\mathcal{G}}$, the variable holding the epoch index of the last synchronization t_s, and the threshold extent ρ. After completion of the initialization, the main training loop starts (Algorithm 1 line 10), which is performed by all actors in parallel. In addition to computing and applying the gradient as an update to the model parameters like in common DFL schemes, each actor keeps track of its accumulated gradient $\mathcal{G}_i^{\Sigma}$ since last synchronization and checks in every iteration if it needs synchronization (Algorithm 1 line 14). If synchronization is required, the actor synchronizes with other actors, resets its state variables, and updates the local model parameters. If none of the actors requires synchronization, the threshold extent is decayed as described in Section 5.3.1. This loop is repeated for the desired number of epochs.

Synchronization Rule As mentioned above, each actor checks in every round if synchronization is required. This check is performed with `RequiresSynchronization`, provided in Algorithm 2. This function verifies whether a given accumulated gradient exceeds the boundaries of the threshold region (defined in Section 5.3.1) and returns the result as `true` or `false`, respectively. To achieve this, the projected distance (Algorithm 2 lines 3–7) and the projection distance (Algorithm 2 lines 8–13) are calculated and compared against the corresponding threshold. Both distances must fall below the threshold value if the accumulated gradient is within the threshold region. Otherwise, the respective actor requires synchronization. As mentioned in Section 5.3.1, the projection distance is calculated using a weighted euclidean distance. The weights for this distance are obtained from the function `ComputeDistanceWeights` in Algorithm 3.

Parameter Synchronization When an actor requires synchronization, the synchronization step is triggered for all actors. We outline the involved operations in Algorithm 4. Initially, each actor exchanges the accumulated gradient with other actors (Algorithm 4 lines 2–3). The synchronization procedure then aggregates the individual accumulated gradients of all actors to obtain a common gradient (Algorithm 4 line 4), which is then applied to the model parameters from the previous synchronization to obtain the synchronized model parameters (Algorithm 4 line 5). Furthermore, this procedure also computes the updated forecast gradient (Algorithm 4 lines 6–10), following the definitions in Section 5.3.1. Together, this procedure results in a global set of model parameters and in an updated forecast gradient, both of which are common among the actors.

Algorithm 1 Gradient Thresholding Protocol

```
Input: number of epochs $\mathcal{T}$, learning rate $\eta$, threshold extent factor $\theta_\rho$, threshold extent decay $\theta_\alpha$
 1: Initialize reference model parameters $\mathcal{M}_R$ to initial state.
 2: for all $i \in \mathcal{N}$ do in parallel
 3:     $\mathcal{M}_i \leftarrow \mathcal{M}_R$                ▷ Initialize model parameters to common initial state.
 4:     $\mathcal{G}_i^\Sigma \leftarrow \mathbf{0}$               ▷ Initialize accumulated gradient to zero gradient.
 5:     $\widetilde{\mathcal{G}} \leftarrow \mathbf{0}$                ▷ Initialize forecast gradient to zero gradient.
 6:     $t_s \leftarrow -1$                  ▷ Initialize epoch of last synchronization.
 7:     $\rho \leftarrow 0$                   ▷ Initialize threshold extent.
 8: end for
 9: for $t = 1$ to $\mathcal{T}$ do
10:     for all $i \in \mathcal{N}$ do in parallel
11:         $\mathcal{G}_i \leftarrow \text{ObtainGradient}(\mathcal{M}_i, \mathcal{D}_i)$
12:         $\mathcal{M}_i \leftarrow \text{ApplyGradient}(\mathcal{M}_i, \mathcal{G}_i, \eta)$
13:         $\mathcal{G}_i^\Sigma \leftarrow \mathcal{G}_i^\Sigma + \mathcal{G}_i$
14:         if $t == 1$ OR RequiresSynchronization $\left(\widetilde{\mathcal{G}}, \mathcal{G}_i^\Sigma, \rho, \theta_\rho\right)$ then    ▷ Alg. 2
15:             $\mathcal{M}_R, \widetilde{\mathcal{G}} \leftarrow \text{Synchronize}(\mathcal{M}_R, \mathcal{G}_i^\Sigma, \widetilde{\mathcal{G}}, \eta)$    ▷ Alg. 4
16:             $\mathcal{M}_i \leftarrow \mathcal{M}_R$
17:             $\mathcal{G}_i^\Sigma \leftarrow \mathbf{0}$           ▷ Reset accumulated gradient.
18:             $\rho \leftarrow \left(1 + \frac{1}{t-t_s}\right)$         ▷ Reset threshold extent.
19:             $t_s \leftarrow t$             ▷ Reset last synchronization epoch.
20:         else
21:             $\rho \leftarrow \theta_\alpha \rho$           ▷ Decay threshold extent.
22:         end if
23:     end for
24: end for
```

Algorithm 2 Gradient Thresholding: Synchronization Rule

Input: forecast gradient $\widetilde{\mathcal{G}}$, local accumulated gradient $\mathcal{G}_i^{\Sigma}$, threshold extent ρ, threshold extent factor θ_ρ

Output: boolean flag indicating if synchronization is needed

```
 1: function RequiresSynchronization($\widetilde{\mathcal{G}}$, $\mathcal{G}_i^{\Sigma}$, $\rho$, $\theta_\rho$)
 2:     $\mathrm{proj}_{[\widetilde{\mathcal{G}}]}\left(\mathcal{G}_i^{\Sigma}\right) \leftarrow \frac{\mathcal{G}_i^{\Sigma} \cdot \widetilde{\mathcal{G}}}{\widetilde{\mathcal{G}} \cdot \widetilde{\mathcal{G}}} \widetilde{\mathcal{G}}$      ▷ Projection onto forecast gradient line.
 3:     $d_{\widetilde{\mathcal{G}}}^{\max} \leftarrow \rho \cdot \left\| \widetilde{\mathcal{G}} \right\|_2$      ▷ Max. distance to forecast gradient.
 4:     $d_{\widetilde{\mathcal{G}}} \leftarrow \left\| \mathrm{proj}_{[\widetilde{\mathcal{G}}]}\left(\mathcal{G}_i^{\Sigma}\right) - \widetilde{\mathcal{G}} \right\|_2$      ▷ Projected distance to forecast gradient.
 5:     if $d_{\widetilde{\mathcal{G}}} > d_{\widetilde{\mathcal{G}}}^{\max}$ then
 6:         return true      ▷ Projected distance exceeds threshold.
 7:     end if

 8:     $W \leftarrow$ ComputeDistanceWeights($\widetilde{\mathcal{G}}$)
 9:     $d_L^{\max} \leftarrow \theta_\rho \cdot \rho \cdot \left\| \widetilde{\mathcal{G}} \right\|_W$      ▷ Max. distance to forecast gradient line.
10:     $d_L \leftarrow \left\| \mathcal{G}_i^{\Sigma} - \mathrm{proj}_{[\widetilde{\mathcal{G}}]}(\mathcal{G}_i^{\Sigma}) \right\|_W$      ▷ Projection distance to forecast gradient line.
11:     if $d_L > d_L^{\max}$ then
12:         return true      ▷ Projection distance exceeds threshold.
13:     end if

14:     return false      ▷ Accumulated gradient is within threshold region.
15: end function
```

Algorithm 3 Gradient Thresholding: Distance Weights

Input: estimated gradient $\widetilde{\mathcal{G}}$

Output: distance weights W

```
 1: function ComputeDistanceWeights($\widetilde{\mathcal{G}}$)
 2:     $W \leftarrow \mathrm{abs}(\widetilde{\mathcal{G}})$
 3:     for $l = 1$ to #layers do      ▷ For each layer.
 4:         for all $w_i \in \mathrm{layer}(l, W)$ do      ▷ For each value of the layer.
 5:             $w_i \leftarrow \max\left(\mathrm{med}(\mathrm{layer}(l, W)), w_i\right)$      ▷ Cap weights by layer median.
 6:         end for
 7:     end for
 8:     $W \leftarrow 1 \,/\, W$
 9:     return $W$
10: end function
```

Algorithm 4 Gradient Thresholding: Synchronization

```
Input: reference model parameters $\mathcal{M}_R$, local accumulated gradient $\mathcal{G}_i^{\Sigma}$, forecast gradient $\widetilde{\mathcal{G}}$, learning rate $\eta$, exponential moving average coefficient $\theta_\beta$
Output: $\mathcal{M}_R$ averaged model parameters, $\widetilde{\mathcal{G}}$ forecast gradient
 1: function SYNCHRONIZE($\mathcal{M}_R$, $\mathcal{G}_i^{\Sigma}$, $\widetilde{\mathcal{G}}$, $\eta$, $\theta_\beta$)
 2:     Send: $\mathcal{G}_i^{\Sigma}$ to other devices
 3:     Receive: $\{\mathcal{G}_j^{\Sigma}\}_{j \in \mathcal{N} \setminus \{i\}}$ from other devices
 4:     $\overline{\mathcal{G}^{\Sigma}} \leftarrow \frac{1}{|\mathcal{N}|} \sum_{j \in \mathcal{N}} \mathcal{G}_j^{\Sigma}$
 5:     $\mathcal{M}_R \leftarrow \text{ApplyGradient}\left(\mathcal{M}_R, \overline{\mathcal{G}^{\Sigma}}, \eta\right)$
 6:     if $\widetilde{\mathcal{G}} == \mathbf{0}$ then
 7:         $\widetilde{\mathcal{G}} \leftarrow \overline{\mathcal{G}^{\Sigma}}$
 8:     else
 9:         $\widetilde{\mathcal{G}} \leftarrow \frac{\left\|\overline{\mathcal{G}^{\Sigma}}\right\|_2}{\left\|\theta_\beta \frac{\overline{\mathcal{G}^{\Sigma}}}{\left\|\overline{\mathcal{G}^{\Sigma}}\right\|_2} + (1-\theta_\beta) \frac{\widetilde{\mathcal{G}}}{\|\widetilde{\mathcal{G}}\|_2}\right\|_2} \left(\theta_\beta \frac{\overline{\mathcal{G}^{\Sigma}}}{\left\|\overline{\mathcal{G}^{\Sigma}}\right\|_2} + (1-\theta_\beta) \frac{\widetilde{\mathcal{G}}}{\|\widetilde{\mathcal{G}}\|_2}\right)$
10:     end if
11:     return $\mathcal{M}_R$, $\widetilde{\mathcal{G}}$
12: end function
```

5.4. Experiments

To verify Gradient Thresholding, we have conducted experiments with several datasets, model architectures, and configurations. In this section, we clarify the common experimental setup (Section 5.4.1) and present the results of our experiments. These experiments include (1) the comparison with baselines (Section 5.4.2), (2) the investigation of the distance to our threshold region (Section 5.4.3), (3) the visualization of the parameter space during training in a simplified scenario (Section 5.4.4), and (4) the assessment of variability across different hyperparameter choices. The implementation for conducting these experiments is publicly available on GitHub[1].

5.4.1. Experimental Setup

Dataset For our experiments, we consider three datasets for classification tasks with different complexities, namely the Iris dataset [165], the Mnist dataset [166], and the cropped version of the Street View House Numbers (SVHN) dataset [157]. We divide the datasets into train, validation, and test set in the following way: We split the Iris dataset randomly with a ratio of 90% : 10% to obtain train and test set, respectively, and sliced another 10% from the train set to obtain the validation set; for the Mnist dataset, we take the train with 60000 records and the test set with 10000 records as provided by TensorFlow [65], and randomly slice another 10% from the train set to obtain the validation set; and for the SVHN dataset, we take the train set with 73257 records and the test set with 26032 records as provided by TensorFlow Datasets [66], and randomly slice 10% from the train set to obtain the validation set. We then partition the train sets using Dirichlet partitioning into the same number of partitions as there are actors. While the resulting partitions are balanced in size, using the Dirichlet partitioning scheme allows us to control the heterogeneity between the partitions through the concentration parameter α from the Dirichlet distribution. Conceptually, α controls the imbalance of categorical labels across partitions, with $\alpha \to \infty$ approximating identical distributions and $\alpha \to 0$ resulting in completely imbalanced distributions. In our experiments, this parameter was set to $\alpha = 0.05$ for the Iris dataset, $\alpha = 0.25$ for the Mnist dataset, and $\alpha = 0.25$ for the SVHN dataset.

Model Aligned with the three datasets, we conduct the experiments with three different model architectures—one for each dataset. For the three-class classification task on the Iris dataset, we construct the neural network model similarly to the tutorial blog post by Jason Browniee [167]. This model consists of a dense layer of neurons with 8 units using the rectified linear unit activation function, followed by a dense layer with 3 units and a softmax activation function to represent the probabilities of the output labels. We provide the model architecture in Table 5.1. For the image classification task on the Mnist dataset and on the SVHN dataset, we construct two different convolutional neural network models. The model architectures are provided in Table 5.2 for Mnist and Table 5.3 for SVHN. Both models take an image as input (grayscale for Mnist and colored for SVHN) and output the probabilities for the class labels. The model architecture for the SVHN dataset was adopted from a notebook on Kaggle by Dimitrios Roussis [168].

[1]Gradient Thresholding GitHub repository: `https://github.com/ywcb00/GradientThresholding/tree/v0.1.1`

Layer Type	Output Shape	# Parameters	Configuration
Input layer	4	0	None
Densely-connected layer	8	40	ReLU activation
Densely-connected layer	3	27	Softmax activation

Table 5.1.: Model architecture of the neural network model for the classification task on the Iris dataset.

Layer type	Output Shape	# Parameters	Configuration
Input layer	26×26	0	None
2D convolution layer	$26 \times 26 \times 32$	320	3×3 kernel ReLU activation
2D max pooling operation	$13 \times 13 \times 32$	0	2×2 pool
2D convolution layer	$11 \times 11 \times 64$	18496	3×3 kernel ReLU activation
2D max pooling operation	$5 \times 5 \times 64$	0	2×2 pool
Input flattening	1600	0	None
Dropout layer	1600	0	50% drop rate
Densely-connected layer	10	16010	Softmax activation

Table 5.2.: Model architecture of the convolutional neural network model for the classification task on the Mnist dataset.

We use the stochastic gradient descent optimizer (SGD) with categorical cross-entropy loss function for the Iris and Mnist model, and adopt the adaptive moment estimation optimizer (ADAM) with sparse categorical cross-entropy loss function for the SVHN dataset. For more detailed information about the model architectures, please refer to the open-source implementation.

General Configuration In addition to the dataset and the model architecture, our experiments rely on the following general configurations:

1. *Number of Epochs*: Throughout the experiments, the total number of epochs to train is set to 100 for the Iris dataset, 20 for the Mnist dataset, and 50 for the SVHN dataset. While these numbers are not tuned to achieve the best performance, we fixed them before conducting the experiments and did not tweak them to improve the results.
2. *Learning Rate*: We set the learning rate of the optimizer to 0.1, 0.2, and 0.0002 for the Iris, Mnist, and SVHN dataset, respectively. These values result from a grid search with various learning rates and are chosen based on the corresponding validation loss. We provide the results of the grid search in Appendix A.1.
3. *Batch Size*: The batch size was set to 32, 256, and 128 for the Iris, Mnist, and SVHN datasets, respectively. Similar to the number of epochs, we determined these values before conducting the experiments and did not change them afterwards.
4. *Number of Actors*: Unless otherwise specified, we conducted the experiments using a network of 8 actors for the Iris dataset and a network of 4 actors for both the Mnist and the SVHN datasets.

Layer type	Output Shape	# Parameters	Configuration
Input layer	$32 \times 32 \times 3$	0	None
2D convolution layer	$32 \times 32 \times 32$	896	3×3 kernel ReLU activation
Batch normalization	$32 \times 32 \times 32$	128	None
2D convolution layer	$32 \times 32 \times 32$	9248	3×3 kernel ReLU activation
2D max pooling operation	$16 \times 16 \times 32$	0	2×2 pool
Dropout layer	$16 \times 16 \times 32$	0	30% drop rate
2D convolution layer	$16 \times 16 \times 64$	18496	3×3 kernel ReLU activation
Batch normalization	$16 \times 16 \times 64$	256	None
2D convolution layer	$16 \times 16 \times 64$	36928	3×3 kernel ReLU activation
2D max pooling operation	$8 \times 8 \times 64$	0	2×2 pool
Dropout layer	$8 \times 8 \times 64$	0	30% drop rate
2D convolution layer	$8 \times 8 \times 128$	73856	3×3 kernel ReLU activation
Batch normalization	$8 \times 8 \times 128$	512	None
2D convolution layer	$8 \times 8 \times 128$	147584	3×3 kernel ReLU activation
2D max pooling operation	$4 \times 4 \times 128$	0	2×2 pool
Dropout layer	$4 \times 4 \times 128$	0	30% drop rate
Input flattening	2048	0	None
Densely-connected layer	128	262272	ReLU activation
Dropout layer	128	0	40% drop rate
Densely-connected layer	10	1290	Softmax activation

Table 5.3.: Model architecture of the convolutional neural network model for the classification task on the SVHN dataset.

5.4.2. Baseline Comparison

Gradient Thresholding tries to optimize the trade-off between communication efficiency and performance. Therefore, the key performance indicator is given by the interaction of the loss and the number of synchronizations. We conducted experiments to compare Gradient Thresholding with the following four baselines:

- *Baseline 1*: Standard DFL with synchronization after every local update.
- *Baseline 2*: Local computation with a low number of 2 local updates per communication round.
- *Baseline 3*: Local computation with a medium number of 5 local updates per communication round.
- *Baseline 4*: Local computation with a high number of local updates per communication round; 10 local updates for the Cifar and SVHN datasets, and 8 local updates for the Mnist dataset.

The hyperparameter θ_ρ that controls the sensitivity of Gradient Thresholding, as described in Section 5.3.1, has been selected considering the validation loss in experiments with various values for θ_ρ. Based on these evaluations, we set the parameter at $\theta_\rho = 2$ for the Iris dataset, $\theta_\rho = 1.75$ for the Mnist dataset, and $\theta_\rho = 2$ for the SVHN dataset. We provide the numeric results of these experiments in Appendix A.2.

We trained the baselines and Gradient Thresholding using the respective hyperparameter θ_ρ, and the configurations described in Section 5.4.1. Since data partitioning and model initialization are random, we performed these experiments using four different seed values. During this process, we recorded the test loss and the number of synchronizations for each approach and each dataset. The results are provided in Table 5.4 for the Iris dataset, Table 5.5 for the Mnist dataset, and Table 5.6 for the SVHN dataset.

Considering the results for the Iris dataset in Table 5.4, Gradient Thresholding clearly outperforms the baselines for seeds 668 and 669. For seed 667, however, Baseline 3 outperforms Gradient Thresholding, as it requires fewer synchronizations to achieve a lower loss. The results with a seed value of 666 do not allow for a definitive comparison, since Gradient Thresholding performs better than Baseline 1 but worse than Baseline 2. In the results for the Mnist dataset in Table 5.5, Gradient Thresholding clearly outperforms the baselines for all seed values. For seed 666, Baseline 3 and Baseline 4 fail to minimize the loss, while Gradient Thresholding achieves a lower loss with fewer synchronizations than Baseline 2. In the results for the SVHN dataset in Table 5.6, Gradient Thresholding achieves better results than the baselines for seed values 666 and 667. The results for seed 668 and seed 669, however, do not allow direct comparisons, as Gradient Thresholding positions itself in-between baselines with both the loss value and the number of synchronizations.

Figure 5.1, Figure 5.2, and Figure 5.3 demonstrate the results for the Iris, Mnist, and SVHN datasets, respectively. To facilitate visual comparison, we present the ratios to the results of Gradient Thresholding for both the loss (y-axis) and the number of synchronizations (x-axis). Thus, all results of Gradient Thresholding are located at position $(1, 1)$ and the values of the baseline experiments show the loss and the number of synchronizations relative to Gradient Thresholding. Making this transformation into ratios also allows us to define all experiments within the unit square from the origin to the results of Gradient Thresholding as clearly outperforming Gradient Thresholding. Although three results from the baseline experiments on the Iris dataset fall within this area, the other baseline results are clearly outside of this unit square area. Furthermore, none of the baseline results clearly outperforms Gradient Thresholding in the experiments on the Mnist and SVHN datasets. Hence, the concept of Gradient Thresholding proves at least equally effective as the baselines. Moreover, the baseline results that show a higher loss and more synchronizations than Gradient Thresholding indicate the superiority of Gradient Thresholding.

In addition to recording the final test loss with the corresponding number of synchronizations, we tracked the test loss during training at each synchronization in Gradient Thresholding. We visualize the results for seed 667 with the SVHN dataset in Figure 5.4. This visualization shows that Gradient Thresholding positions itself in-between Baseline 2 and Baseline 3 in terms of the number of synchronizations, while achieving the best loss among all experiments. Additional visualizations of this type for the different datasets and seed values are provided in Appendix A.3.

Exp.	SEED 666		SEED 667		SEED 668		SEED 669	
	Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
BL1	0.539029	100	0.501504	100	0.474516	100	0.487596	100
BL2	0.515485	51	0.563500	51	0.508835	51	0.450143	51
BL3	0.516858	21	0.556961	21	0.609173	21	0.481423	21
BL4	0.578848	11	0.681120	11	0.776382	11	0.523183	11
GT	0.526247	92	0.587449	28	0.473216	90	0.385178	90

Table 5.4.: Test loss and number of synchronizations ('Sync.') for the baseline (BL) and Gradient Thresholding (GT) experiments ('Exp.') on the Iris dataset over different seeds. Gradient Thresholding clearly outperforms the baselines in the cases shaded in green, while it is inferior to the baselines in the case shaded in red.

Exp.	SEED 666		SEED 667		SEED 668		SEED 669	
	Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
BL1	0.232185	20	0.285152	20	0.205982	20	0.497485	20
BL2	0.386331	11	0.296058	11	0.206932	11	0.458328	11
BL3	2.451042	5	0.291849	5	0.246449	5	0.542560	5
BL4	2.435904	4	0.331246	4	0.258262	4	0.599970	4
GT	0.359094	8	0.319479	4	0.231651	5	0.545842	4

Table 5.5.: Test loss and number of synchronizations ('Sync.') for the baseline (BL) and Gradient Thresholding (GT) experiments ('Exp.') on the Mnist dataset over different seeds. Gradient Thresholding clearly outperforms the baselines for all seed values.

Exp.	SEED 666		SEED 667		SEED 668		SEED 669	
	Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
BL1	0.426264	50	0.417600	50	0.378228	50	0.395136	50
BL2	0.437644	26	0.383815	26	0.378151	26	0.373628	26
BL3	0.418537	11	0.365922	11	0.425989	11	0.427693	11
BL4	0.473795	6	0.401534	6	0.479799	6	0.505273	6
GT	0.417503	31	0.358765	16	0.380986	23	0.374048	21

Table 5.6.: Test loss and number of synchronizations ('Sync.') for the baselines (BL) and Gradient Thresholding (GT) experiments ('Exp.') on the SVHN dataset over different seeds. Gradient Thresholding clearly outperforms the baselines in the cases shaded in green, while the unshaded cases do not provide clear indications of the superiority.

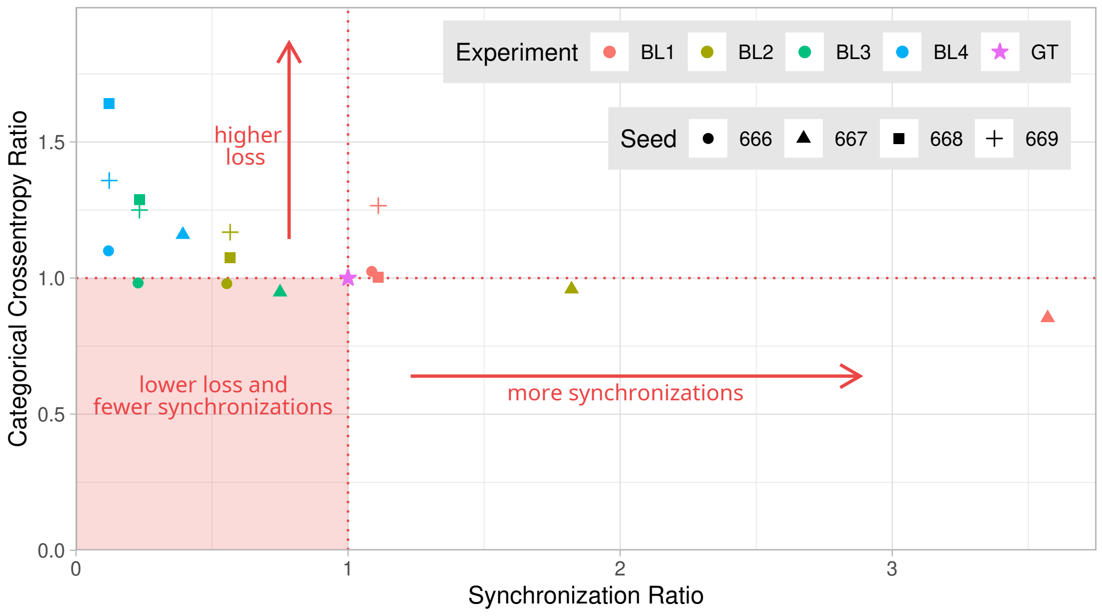


Figure 5.1.: Comparison of the experimental results on the Iris dataset of Gradient Thresholding (GT) to the baselines (BL) over multiple seeds. Both axes are scaled relative to the result of Gradient Thresholding. The x-axis shows the relative number of synchronizations and the y-axis shows the relative test loss. The unit square from the origin to $(1, 1)$ (shaded in red) marks the area in which the baselines definitely outperform Gradient Thresholding.

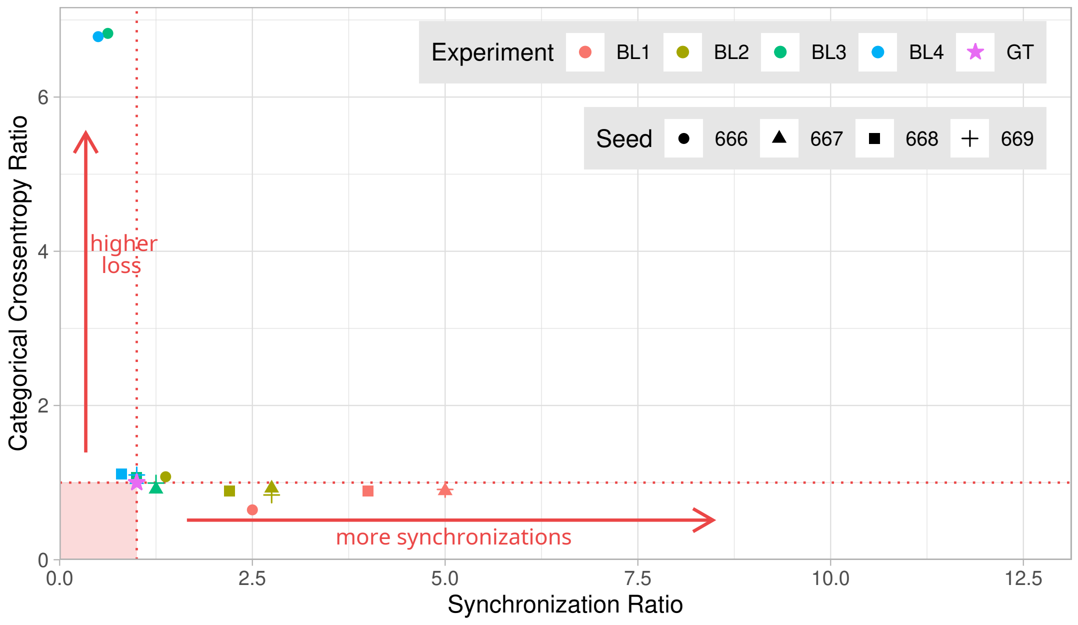


Figure 5.2.: Comparison of the experimental results on the Mnist dataset of Gradient Thresholding (GT) to the baselines (BL) over multiple seeds. Both axes are scaled relative to the result of Gradient Thresholding. The representation of visualized elements is identical to Figure 5.1.

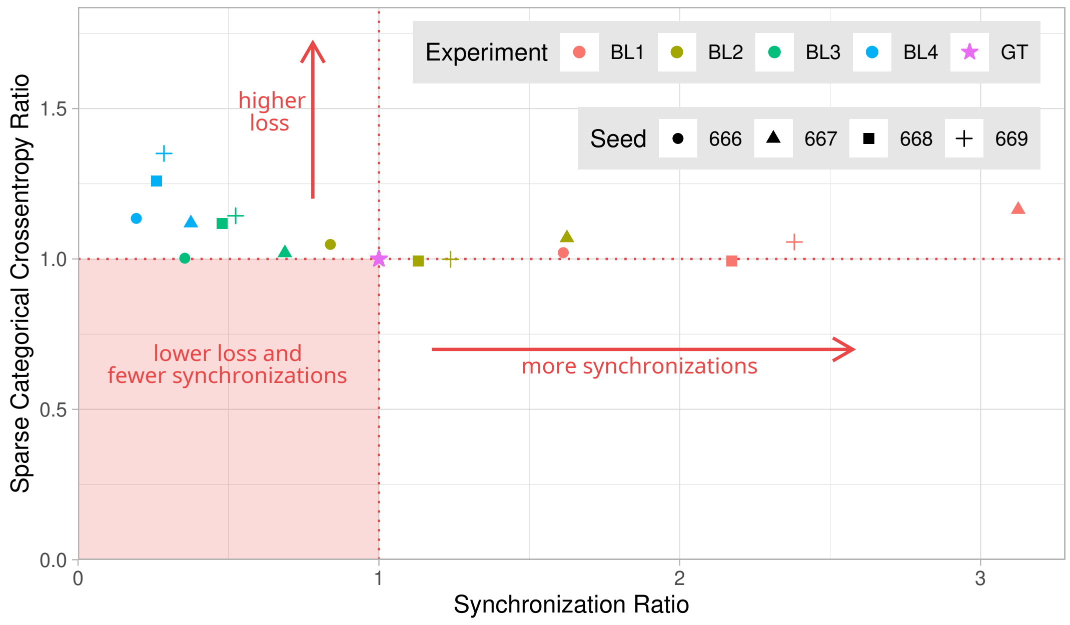


Figure 5.3.: Comparison of the experimental results on the SVHN dataset of Gradient Thresholding (GT) to the baselines (BL) over multiple seeds. Both axes are scaled relative to the result of Gradient Thresholding. The representation of visualized elements is identical to Figure 5.1.

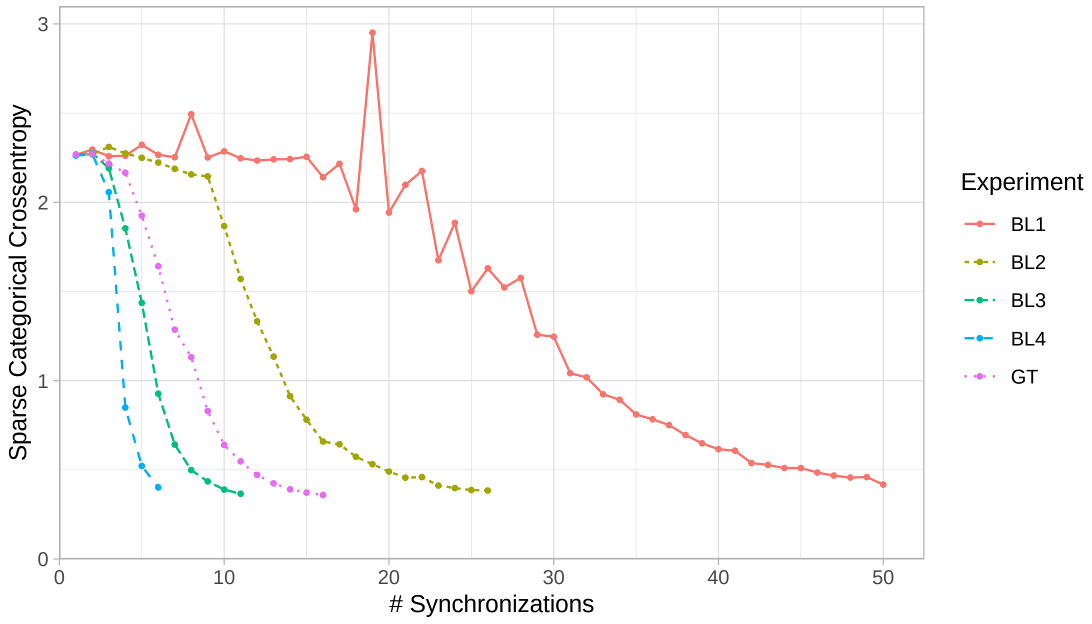


Figure 5.4.: Test loss of experiments on the SVHN dataset with seed value 667 over the number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

5.4.3. Threshold Distance

In the experiment on Gradient Thresholding in Section 5.4.2, we recorded the distances in our threshold region and the values of the two threshold components that characterize the region, as explained in Section 5.3.1. Figure 5.5 and Figure 5.7 show the projected distances of the individual actors with seed value 667 for the Iris dataset and the SVHN dataset, respectively, and Figure 5.6 and Figure 5.8 show the projection distances. Figure 5.5 clearly depicts that the local model parameters of the actors evolve towards the threshold center, as the distance decreases. Often, the actors even pass the threshold center, indicated by the consecutive increase in distance. This scenario is also visible in Figure 5.7. Between the second and third synchronization points, for example, Actor 3 first approximates the threshold center before distancing itself again. Figure 5.8 depicts the projection distances, revealing diverse rates of deviation from the forecast gradient line among actors. It demonstrates the progression of the actors' model parameters towards the lateral threshold boundary (dashed line) until an actor exceeds the boundary and sparks the synchronization between the actors (dotted vertical line). Figure 5.6 shows the sensitivity of our threshold region to slowly-changing dimensions of the model parameters, as the high peaks in distance are caused by heavily weighted dimensions in the weighted distance. The synchronization points in all four figures demonstrate the ability to train multiple consecutive rounds without synchronizing while staying inside our threshold region. We provide identical visualizations for the Mnist dataset with seed value 667 as well as for all datasets with seed value 666 in Appendix A.4.

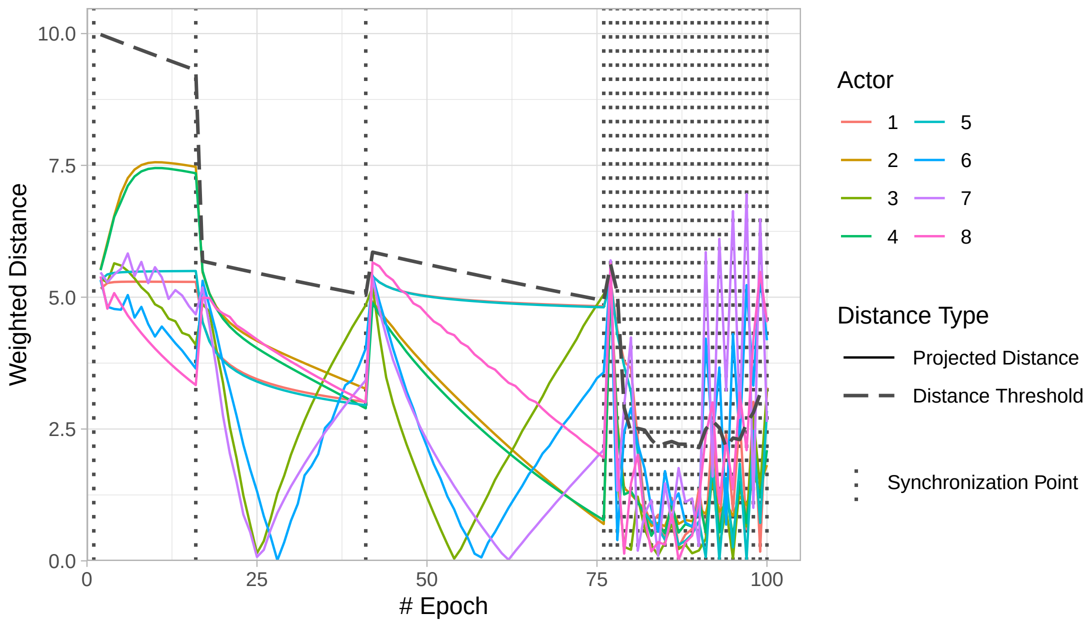


Figure 5.5.: Projected distances of the individual actors in the experiment on the Iris dataset with a seed value of 667. The dashed line represents the corresponding threshold boundary, which triggers a synchronization when exceeded. Synchronization points are indicated by dotted vertical lines. Starting from epoch 76, the model parameters reach a state in which the actors constantly diverge from each other and, thus, trigger synchronization in every round due to exceeding the threshold of the projected distance or the projection distance (see Figure 5.6).

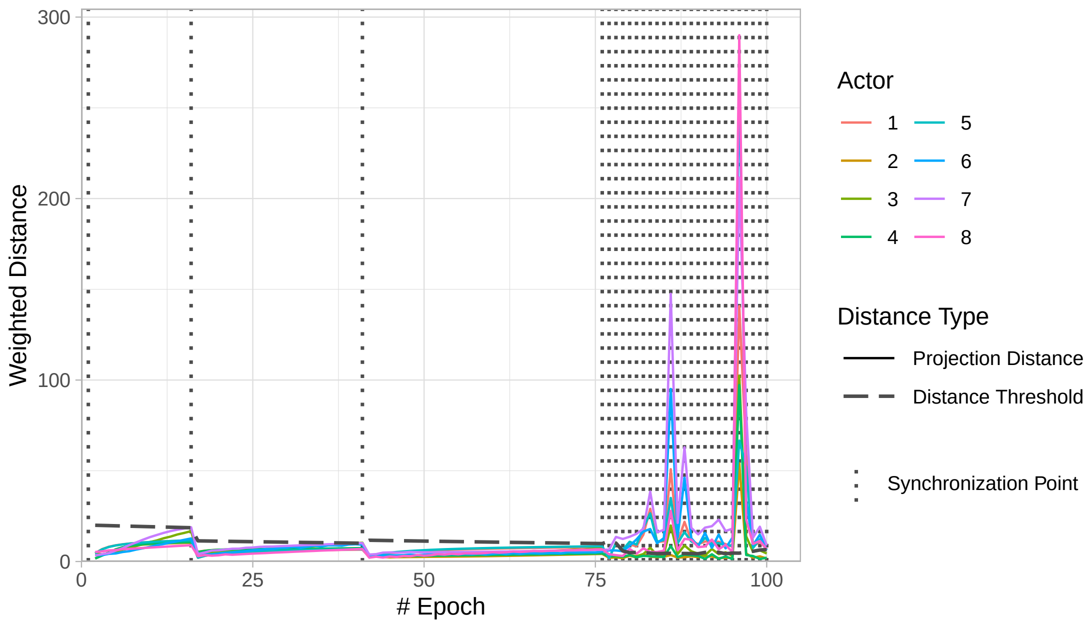


Figure 5.6.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) under the same setting as in Figure 5.5.

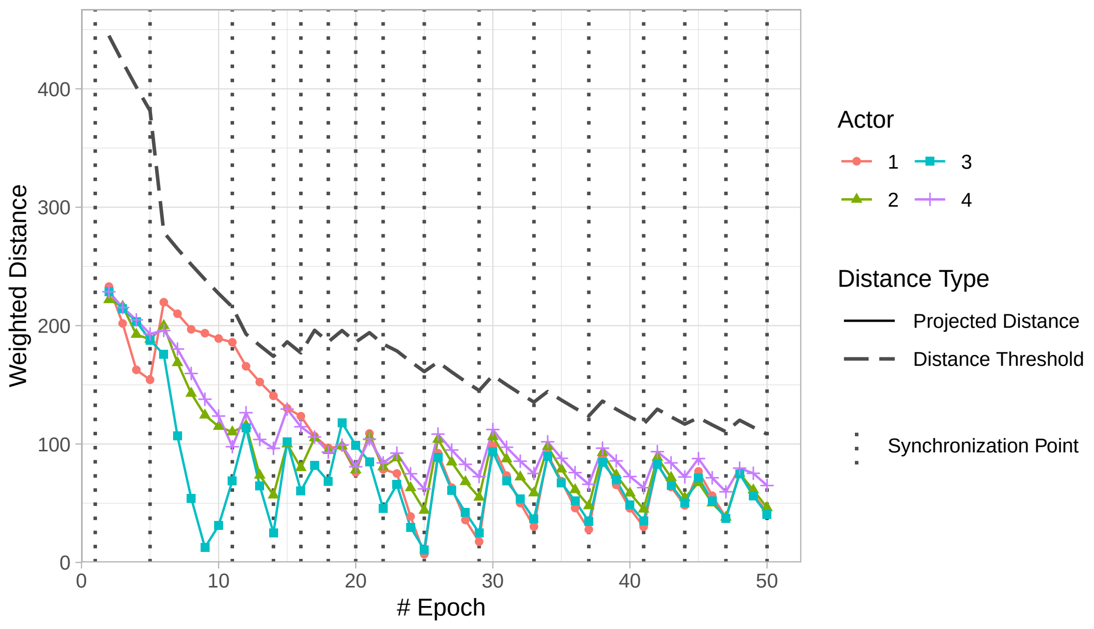


Figure 5.7.: Projected distances of the individual actors in the experiment on the SVHN dataset with a seed value of 667. The dashed line represents the corresponding threshold boundary, which triggers a synchronization when exceeded. Synchronization points are indicated by dotted vertical lines.

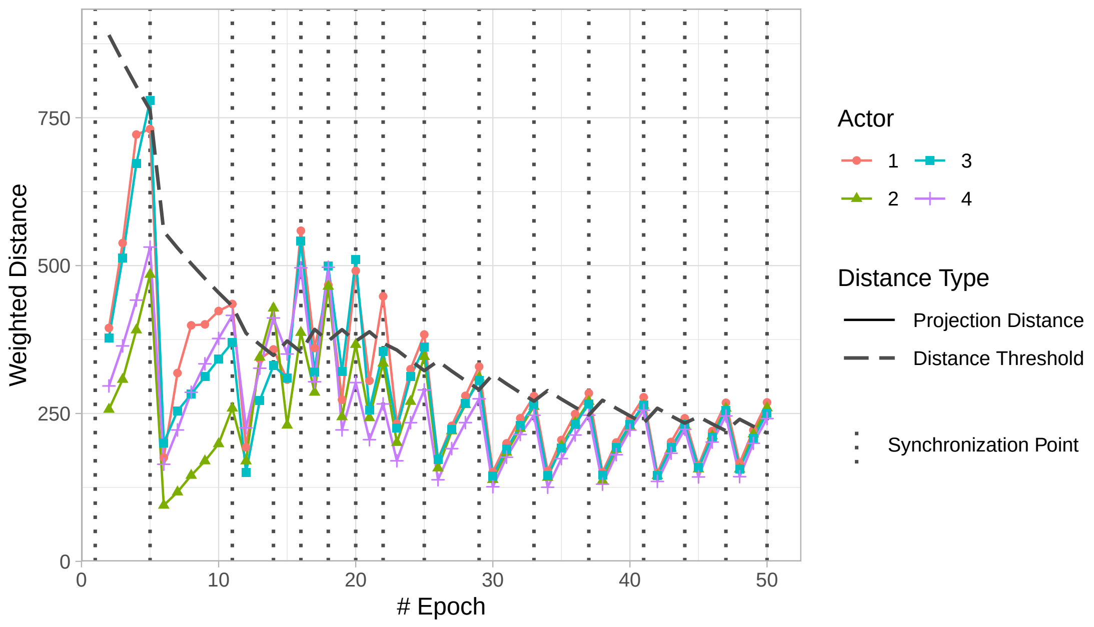


Figure 5.8.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) under the same setting as in Figure 5.8.

5.4.4. Parameter Space Illustration

To illustrate the concept of Gradient Thresholding more clearly, we performed an experiment on a simpler dataset with a model that has only two parameters. This enables to visualize the parameter space in two-dimensional space and, thus, opens the possibility to plot the model parameters. In this experiment, we used the Salary dataset [169], which is known for demonstrating simple linear regression. This dataset consists of 30 records with two features, namely salary as the target feature and years of experience as the explanatory feature. We then construct a neural network that accepts the years of experience as input and passes it to a single neuron with a linear activation function in order to come up with the corresponding salary forecast. This results in a total of two parameters, representing the intercept and the slope similar to a simple linear regression model and allowing for interpretation of the parameters. We leverage the stochastic gradient descent optimizer (SGD) to train the model. Since this experiment serves only for illustration purposes, we do not aim at finding the optimal point. Instead, we optimize the model with a significantly low learning rate so that the model parameters approach the optimum slowly. With this setup, we run the learning procedure on four actors for 15 epochs using Gradient Thresholding, recording the state of the parameters at every epoch. We then visualize the developing parameters of the individual actors by plotting their state in the parameter space. We present these graphs for epochs 2, 6, 12, and 15 in Figure 5.9, Figure 5.10, Figure 5.11, and Figure 5.12, respectively.

Since the first epoch always synchronized the model parameters, the threshold region of Gradient Thresholding is established and applied starting with epoch 2. Figure 5.9 shows how the individual actors train their model parameters towards the forecast parameters, starting from their common synchronized parameters from epoch 1. As the model parameters do not exceed the threshold boundary, there is no need for synchronization in epoch 2. This is the case until epoch 6, as shown in Figure 5.10. Here, actor 1 and actor 2 exceed the threshold limits in their projected distance, triggering the synchronization in epoch 6. Based on this synchronization, a new threshold region is established. This region allows for multiple epochs without synchronizing again, until the next transcendence of the threshold boundary. In contrast to epoch 6, this time the model parameters did not develop through the whole threshold region and surpass it on the opposite side. The parameters developed slower than the decay of the threshold region and, therefore, did not manage to remain inside the threshold, as seen in Figure 5.11. Subsequently, the actors synchronize with each other and update the threshold region, which remains valid until the end of training in epoch 15. The last epoch of a training procedure synchronizes the model parameters again, resulting in the final parameter state of epoch 15 shown in Figure 5.12.

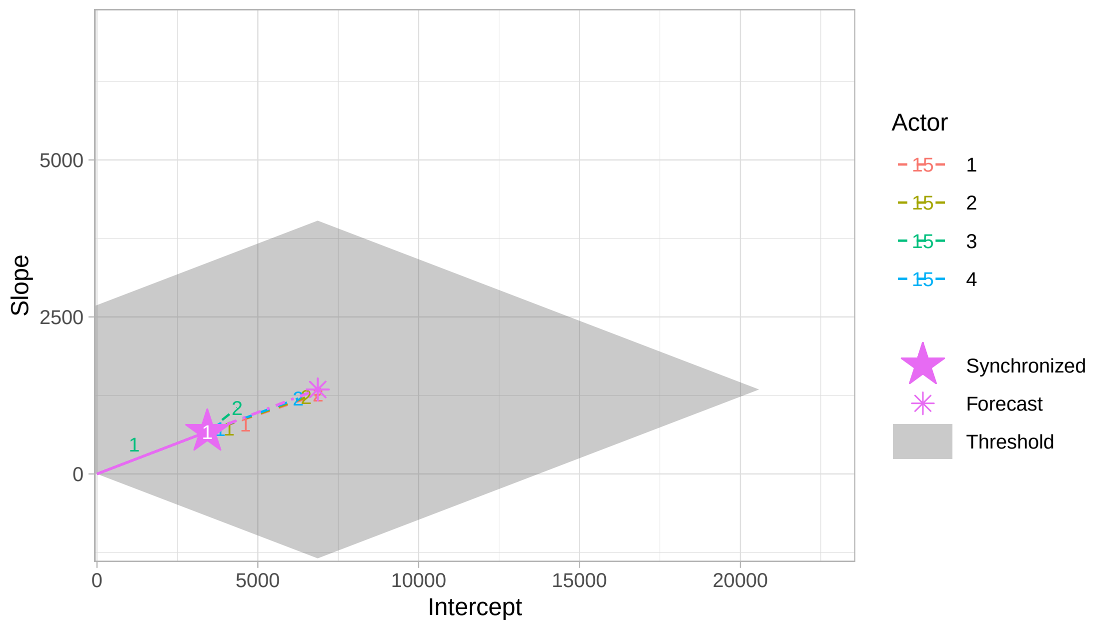


Figure 5.9.: Model parameters of the individual actors after epoch 2 of the training process under the simplified model and dataset, together with the synchronized model parameters from the synchronization points, the forecast parameters advanced by the gradient forecast, and the threshold region that surrounds the forecast parameters. Numeric labels in the plot indicate the epoch of the respective parameters.

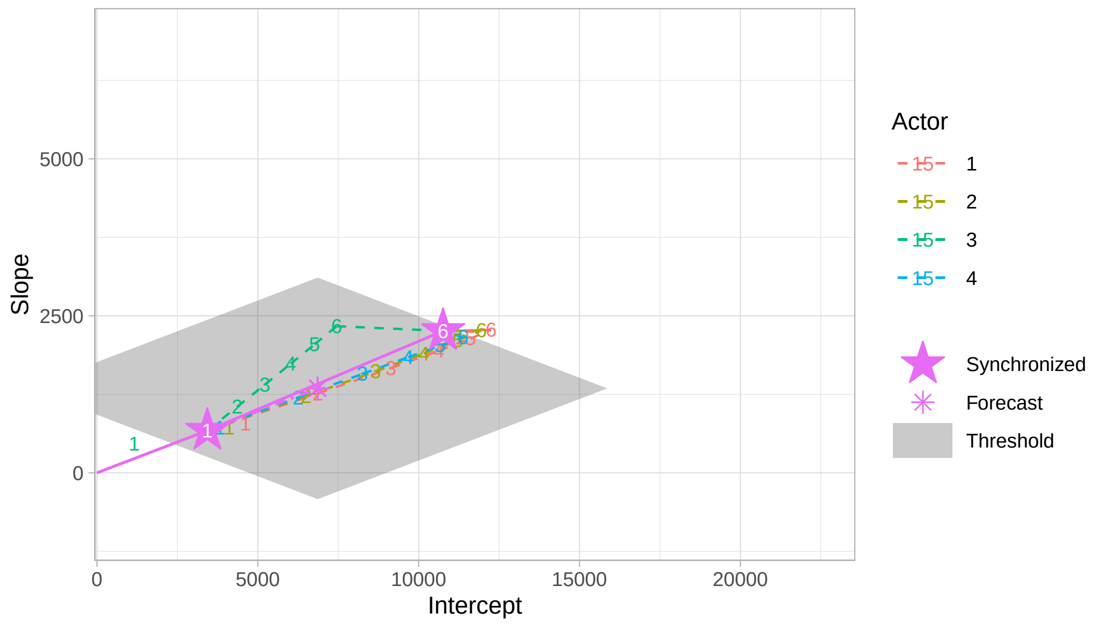


Figure 5.10.: Model parameters after epoch 6 with identical representations as in Figure 5.9. Actor 1 and actor 2 exceed the threshold region in epoch 6, triggering the synchronization between actors and resulting in new synchronized model parameters.

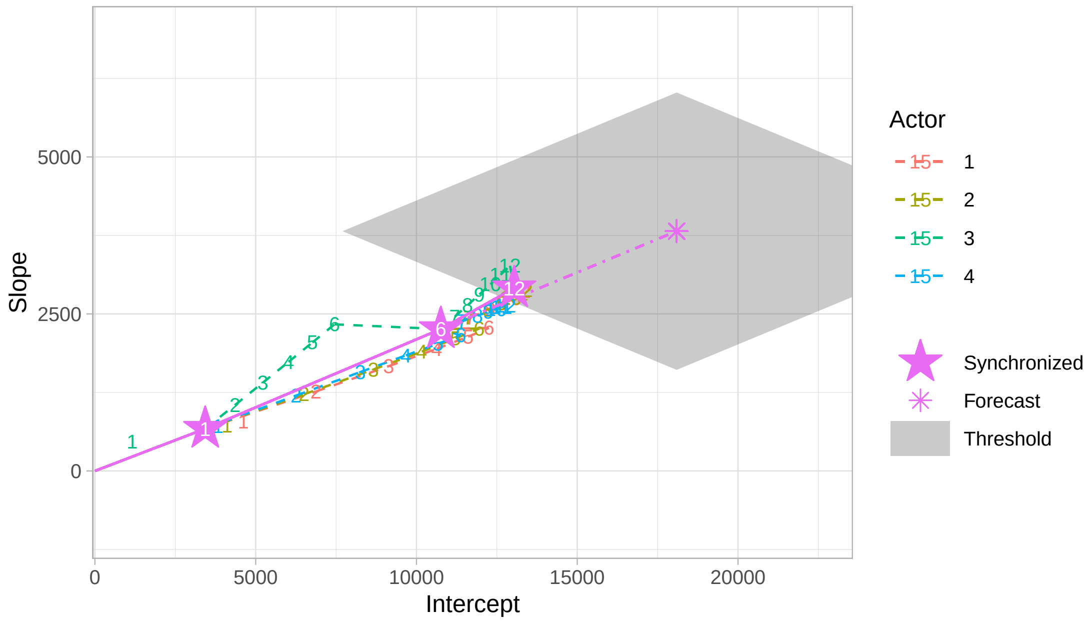


Figure 5.11.: Model parameters after epoch 12 with identical representations as in Figure 5.9. Note that the threshold region surrounds the forecast parameters from the last synchronization. Actor 4 exceeds the threshold region in epoch 12 due to the threshold decay, resulting in new synchronized model parameters.

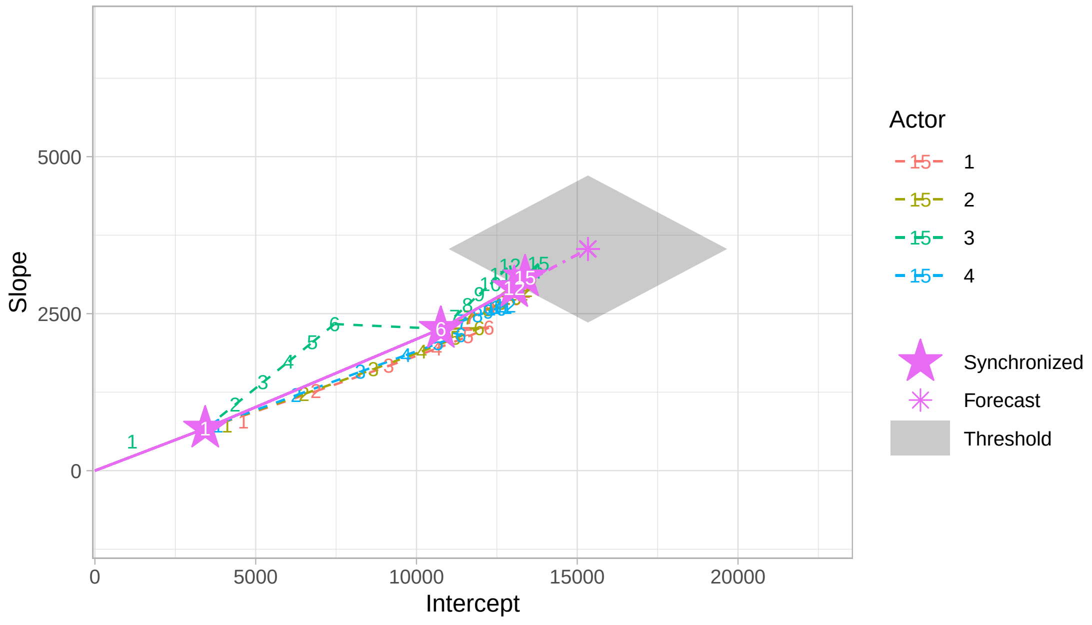


Figure 5.12.: Model parameters after completion of the training process in epoch 15, showing the entire path of synchronized model parameters in the parameter space. Actors synchronize their model parameters in the final epoch by default, yielding the final synchronized gradient. The illustrated components are visualized identical to Figure 5.9.

5.4.5. Variability Test

In Section 5.4.2 and Section 5.4.3, we conducted experiments using fixed values for the number of actors, the Dirichlet partitioning parameter α, and the learning rate. Here, we examine the variability of the results for different values for these parameters. We consider the SVHN dataset and the corresponding model as described in Section 5.4.1. We then perform experiments using Gradient Thresholding with $\theta_\rho = 2$, while varying the number of actors, the Dirichlet parameter α, and the learning rates. Note that we only change one parameter at a time and, thus, set the other parameters to 4 actors, $\alpha = 0.25$, and a learning rate of 0.0002 when they are not under test. These fixed parameters are equal to the experiments in Section 5.4.2 and Section 5.4.3. The resulting test loss for the experiments with seed value 13 is presented in Figure 5.13, Figure 5.14, and Figure 5.15 for different numbers of actors, various Dirichlet partitioning parameters α, and varying learning rates, respectively. We also provide the results for the experiments with seed value 14 and seed value 15 in Appendix A.5.

The test losses for different numbers of actors (Figure 5.13) indicate that the optimization problem becomes more difficult with more actors. This is justified by the fact that the size of the total dataset stays the same, thus an actor in the experiment with 32 actors holds eight times less data than an actor in the experiment with 4 actors. Due to this greater complexity, the number of synchronizations and the resulting loss increase with more actors. The experiment with 2 actors exhibits strong fluctuations in loss, which is due to random partitioning, as confirmed by the results of the same experiment with seed 14 (Figure A.23).

Experiments with different parameters α for Dirichlet partitioning yield exactly the results that we expect, the number of synchronizations conducted by Gradient Thresholding aligns with the level of heterogeneity among the data distributions at the actors. This is supported by Figure 5.14 and is perfectly in agreement with the concept of Gradient Thresholding. The experiment with the highest parameter for α (i.e., the lowest data heterogeneity) reaches the lowest loss value with the fewest synchronizations among the experiments. This interactive trend is followed by all values of α down to the experiment with $\alpha = 0.1$, which achieves the highest final loss of all experiments after a total of 38 synchronizations.

The results for different learning rates are also in line with our expectations, as well as with the concept of Gradient Thresholding. As presented in Figure 5.15, lower learning rates demand fewer synchronizations between actors. This behavior is reasonable, since the model parameters at the actors evolve slower and, hence, do not diverge as fast as they would with higher learning rates. However, slower development also implies slower approximation of the optimal model parameters, as clearly evident from the loss values in Figure 5.15. Therefore, higher learning rates—as long as they are not too large—can achieve better loss but raise the number of synchronizations. If the learning rate is too high, the learning process faces problems with converging in general, as confirmed by the experiment with the highest learning rate in Figure 5.15.

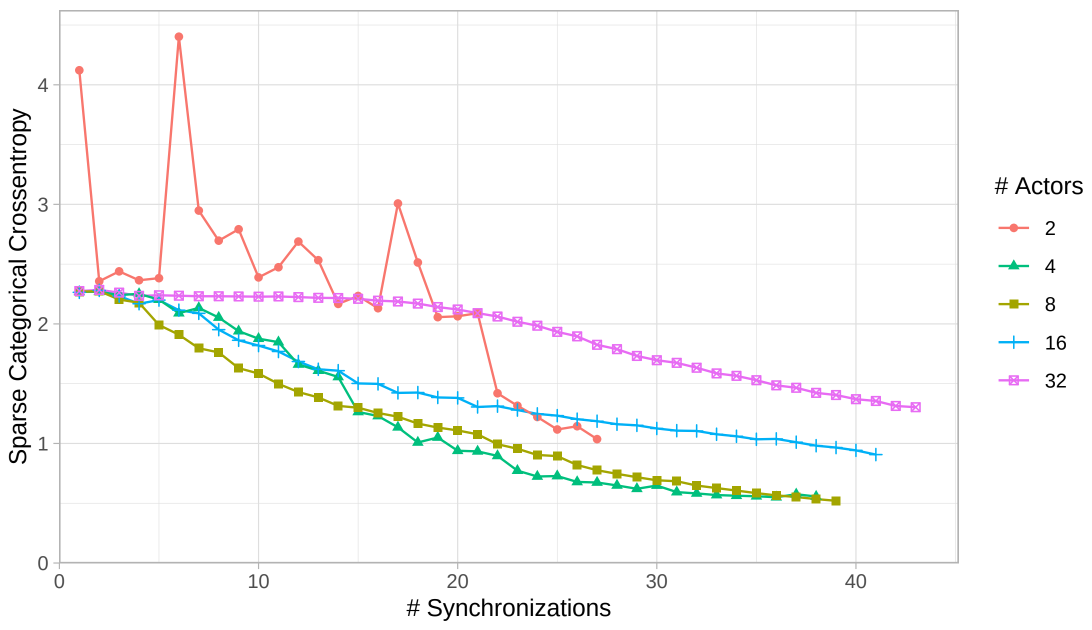


Figure 5.13.: Test loss of experiments on the SVHN dataset with seed 13 for various numbers of actors over the number of synchronizations.

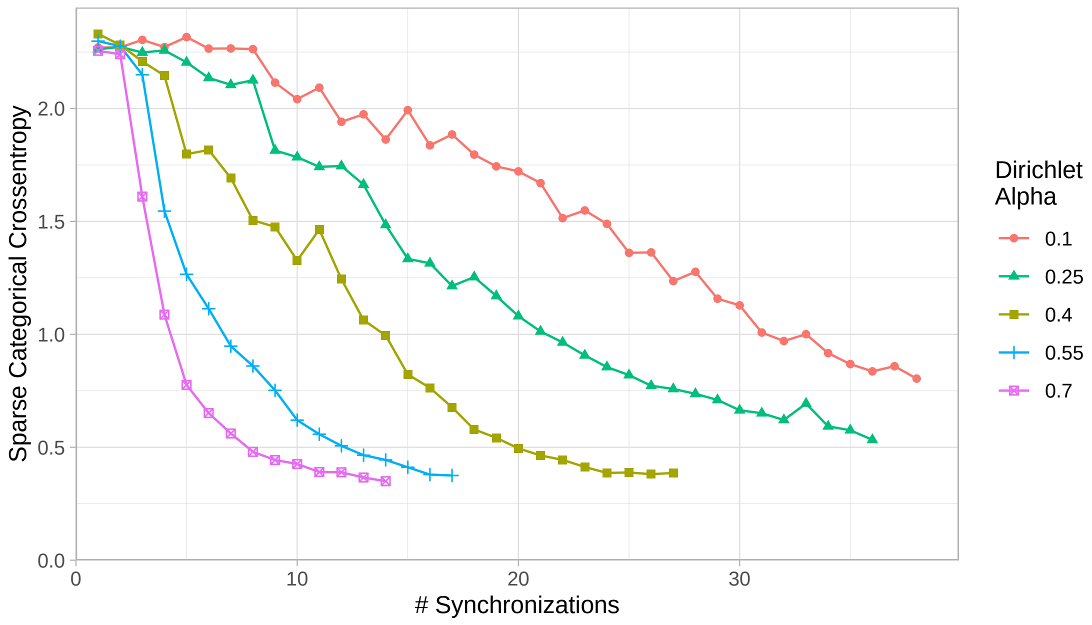


Figure 5.14.: Test loss of experiments on the SVHN dataset with seed 13 for various values for the Dirichlet partitioning parameter α over the number of synchronizations.

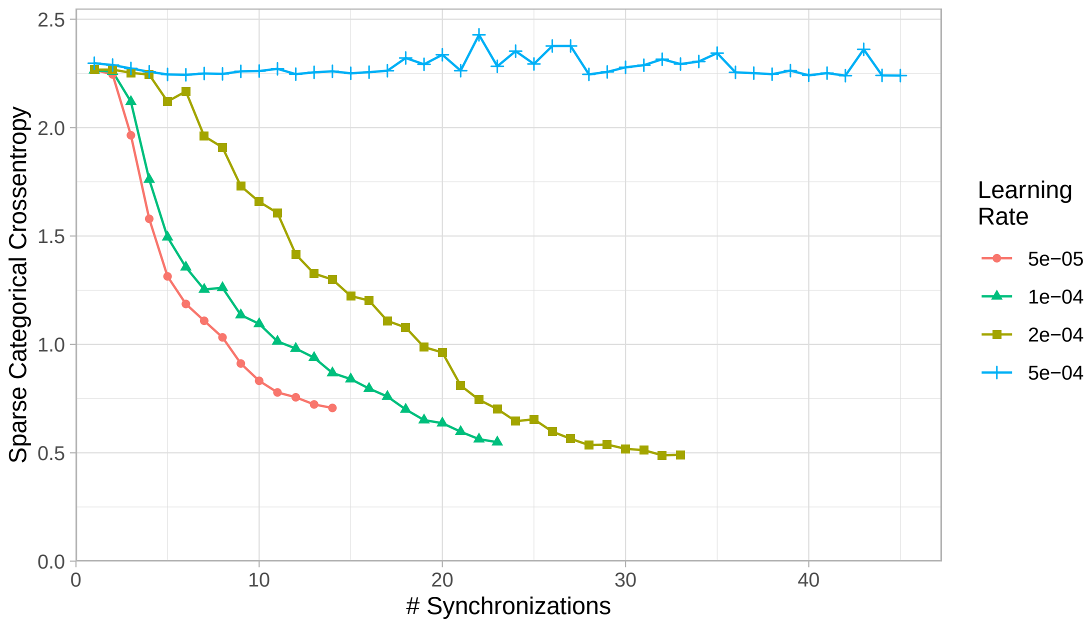


Figure 5.15.: Test loss of experiments on the SVHN dataset with seed 13 for various learning rates over the number of synchronizations.

5.5. Discussion and Future Work

In this chapter, we have explained the methodology of Gradient Thresholding and outlined its general algorithm. Our experimental findings confirm the concept of Gradient Thresholding and its superiority over baselines. However, Gradient Thresholding is based on a speculative forecast of the successive gradient and, therefore, does not provide theoretical guarantees regarding its applicability. Nevertheless, we can discuss the behavior of Gradient Thresholding in two border cases:

1. If an actor advances the local model parameters beyond the threshold region in every round, the actors synchronize their model parameters in each round. This scenario increases the communication cost drastically, rendering Gradient Thresholding equivalent to DFL training without communication-efficient techniques like local computation.
2. If the threshold region is constructed to cover an excessive section of the parameter space, the actors might train for a vast number of epochs without synchronizing. The fact that the threshold region decreases in each epoch ensures that an actor will eventually exceed the threshold. While this does not incur additional communication cost, it wastes time and computing resources by performing redundant training epochs.

Although we compare Gradient Thresholding with baselines that use local computation strategies, it should not be regarded as a dynamic local computation strategy. If the model parameters of an actor exceed the threshold region, the respective actor triggers synchronization for all participating actors. Hence, Gradient Thresholding differs from dynamic strategies for local computation. Furthermore, it can be combined with local computation by performing multiple local model updates between the verifications of whether an actor requires synchronization.

While the concept of Gradient Thresholding is proving advantageous, there are still numerous opportunities for future improvements. Here, we list some promising avenues for future work:

- *Theoretical foundation*: The threshold region and the associated distance weights rest solely on our humble intuition. Although our intuition appears to yield considerably good results, Gradient Thresholding requires a theoretical basis to enable general validity. This includes revising the distance weights so that they reflect a statistically plausible principle.
- *Adaptation of sensitivity*: Gradient Thresholding relies on the hyperparameter θ_ρ that controls the extent of the threshold region. The choice of this hyperparameter depends on the data situation and the model. In our experiments, we set θ_ρ to a single, fixed value based on evaluations on validation data over various seed values. Since the seed value also affects Dirichlet data partitioning, however, we have utilized the same hyperparameter θ_ρ for different data situations, which does not yield the actual optimal choice. Thus, Gradient Thresholding would benefit from sophisticated strategies to dynamically adjust the threshold extent during training. Since the threshold extent is established globally and, thus, unaware of the data itself, the adjustment is limited to the information exchanged between actors at synchronization points. Therefore, a potential future avenue is to model the synchronized gradients through a stochastic random walk model, allowing the

adaptation of the threshold extent based on the empirical variance under the respective model.

- *Arbitrary network topology*: We have considered the fully-connected network topology for designing Gradient Thresholding. This offers the advantage that all actors share a common threshold region. However, DFL systems often show arbitrary network topologies, rendering Gradient Thresholding unsuitable. Therefore, future work includes supporting arbitrary network topologies. This can be achieved by maintaining individual threshold regions on every actor. Here, the threshold regions are established on a personalized level, taking into account only actors that are connected to the respective actor. This, in turn, raises challenges regarding threshold boundaries, partial synchronization, and weakly connected actors, which have to be addressed.

6. Conclusions

This thesis outlined our entire journey from DFL fundamentals to the challenge of communication efficiency in DFL. To introduce the field of DFL and define the overall setting, we started by explaining the general paradigm, identifying key research challenges, and discussing popular application scenarios. The multitude of methods found to address different challenges reveals the difficulty in comparing the methods with each other. To facilitate their experimental evaluation within a comparable, common infrastructure, we implemented a modular DFL framework, called MoDeFL. Modules represent the components of a DFL system and are completely configurable in their strategy. Our insight that modules can be strictly separated from each other, together with the straightforward expandability of the module implementations, promotes the use of MoDeFL in research on novel DFL methods. The integration of various communication-efficient techniques and the support of communication-relevant metrics in MoDeFL sparked our interest in the challenge of communication efficiency. Therefore, we surveyed the recent literature on DFL with a particular focus on communication efficiency. Besides establishing a taxonomy for individual strategies and examining the involved communication cost, we identified under-researched directions regarding individual strategies and their combinations. We found dynamic synchronization strategies among these directions, which led us to designing a novel synchronization rule, called Gradient Thresholding. Recognizing that synchronization between actors is the source of communication cost, we try to optimize the trade-off between the number of synchronizations and the performance. Gradient Thresholding therefore speculatively determines a region in the parameter space in which actors are allowed to update their model parameters without synchronizing with each other. This region gives actors enough leeway to omit synchronizations by training multiple consecutive epochs, while the restrictions on the trainable parameter space prevent actors from diverging. In addition to explaining the methodology and providing the algorithm, we further illustrate the concept of Gradient Thresholding in an empirical toy example. The results of our experiments demonstrate the superiority of Gradient Thresholding over baseline experiments in optimizing the trade-off between loss and the number of synchronizations. Furthermore, the results indicate that the careful selection of synchronization points based on gradient properties offers great advantages. Yet, Gradient Thresholding leaves numerous avenues for future research, including exploring its efficacy in other problem settings and improving the adaptation mechanisms to further reinforce its concept and attain optimal convergence performance with minimal communication.

Bibliography

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *International Conference on Artificial Intelligence and Statistics*, 2016. [Online]. Available: `https://api.semanticscholar.org/CorpusID:14955348`.

[2] D. Weissteiner, L. Demelius, and A. Trügler, "Communication efficiency in decentralized federated learning: A survey," *Artificial Intelligence Review*, under review.

[3] K. Bonawitz, H. Eichner, W. Grieskamp, *et al.*, "Towards federated learning at scale: System design," *ArXiv*, vol. abs/1902.01046, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:59599820`.

[4] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, pp. 50–60, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:201126242`.

[5] P. Kairouz, H. B. McMahan, B. Avent, *et al.*, "Advances and open problems in federated learning," *Found. Trends Mach. Learn.*, vol. 14, pp. 1–210, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:209202606`.

[6] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Neural Information Processing Systems*, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:195316471`.

[7] B. Hitaj, G. Ateniese, and F. Pérez-Cruz, "Deep models under the gan: Information leakage from collaborative deep learning," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5051282`.

[8] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients - how easy is it to break privacy in federated learning?" *ArXiv*, vol. abs/2003.14053, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:214728347`.

[9] K. Wei, J. Li, M. Ding, *et al.*, "Federated learning with differential privacy: Algorithms and performance analysis," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3454–3469, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:207847853`.

[10] N. M. Hijazi, M. Aloqaily, M. Guizani, B. Ouni, and F. Karray, "Secure federated learning with fully homomorphic encryption for iot communications," *IEEE Internet of Things Journal*, vol. 11, pp. 4289–4300, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:260674251`.

[11] L. Zhang, J. Xu, P. Vijayakumar, P. K. Sharma, and U. Ghosh, "Homomorphic encryption-based privacy-preserving federated learning in iot-enabled healthcare system," *IEEE Transactions on Network Science and Engineering*, vol. 10, pp. 2864–2880, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:250196012`.

[12] C. Chen, J. Liu, H. Tan, *et al.*, "Trustworthy federated learning: Privacy, security, and beyond," *ArXiv*, vol. abs/2411.01583, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:273811542`.

[13] A. Z. Tan, H. Yu, L.-z. Cui, and Q. Yang, "Towards personalized federated learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, pp. 9587–9603, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:232076330`.

[14] C. Xu, Y. Qu, Y. Xiang, and L. Gao, "Asynchronous federated learning on heterogeneous devices: A survey," *Comput. Sci. Rev.*, vol. 50, p. 100 595, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:237454599`.

[15] Z. Zhao, Y. Mao, Y. Liu, *et al.*, "Towards efficient communications in federated learning: A contemporary survey," *ArXiv*, vol. abs/2208.01200, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:251252933`.

[16] O. Shahid, S. Pouriyeh, R. M. Parizi, Q. Z. Sheng, G. Srivastava, and L. Zhao, "Communication efficiency in federated learning: Achievements and challenges," *ArXiv*, vol. abs/2107.10996, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:236318520`.

[17] O. R. A. Almanifi, C. O. Chow, M.-L. Tham, J. H. Chuah, and J. Kanesan, "Communication and computation efficiency in federated learning: A survey," *Internet Things*, vol. 22, p. 100 742, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:257370398`.

[18] E. Hallaji, R. Razavi-Far, M. Saif, B. Wang, and Q. Yang, "Decentralized federated learning: A survey on security and privacy," *IEEE Transactions on Big Data*, vol. 10, pp. 194–213, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:267335371`.

[19] M. Fang, Z. Zhang, Hairi, *et al.*, "Byzantine-robust decentralized federated learning," in *Conference on Computer and Communications Security*, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:270559900`.

[20] A. C.-C. Yao, "Protocols for secure computations," *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 160–164, 1982. [Online]. Available: `https://api.semanticscholar.org/CorpusID:62613325`.

[21] R. Kanagavelu, Q. Wei, Z. Li, *et al.*, "Ce-fed: Communication efficient multiparty computation enabled federated learning," *Array*, vol. 15, p. 100 207, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:249861304`.

[22] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith, "Calibrating noise to sensitivity in private data analysis," *J. Priv. Confidentiality*, vol. 7, pp. 17–51, 2006. [Online]. Available: `https://api.semanticscholar.org/CorpusID:2468323`.

[23] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, pp. 211–407, 2014. [Online]. Available: `https://api.semanticscholar.org/CorpusID:207178262`.

[24] E. Cyffers and A. Bellet, "Privacy amplification by decentralization," *ArXiv*, vol. abs/2012.05326, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:228083932`.

[25] E. Cyffers, M. Even, A. Bellet, and L. Massouli'e, "Muffliato: Peer-to-peer privacy amplification for decentralized optimization and averaging," *ArXiv*, vol. abs/2206.05091, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:249605841`.

[26] N. M. Hijazi, M. Aloqaily, and M. Guizani, "Collaborative iot learning with secure peer-to-peer federated approach," *Computer Communications*, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:272626777`.

[27] H. Wang, L. Muñoz-González, D. Eklund, and S. Raza, "Non-iid data re-balancing at iot edge with peer-to-peer federated learning for anomaly detection," *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:235628405`.

[28] M. Chen, Y. Xu, H. Xu, and L. Huang, "Enhancing decentralized federated learning for non-iid data on heterogeneous devices," *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 2289–2302, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:260172527`.

[29] C. Li, G. Li, and P. K. Varshney, "Decentralized federated learning via mutual knowledge transfer," *IEEE Internet of Things Journal*, vol. 9, pp. 1136–1147, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:229371278`.

[30] B. Soltani, V. Haghighi, Y. Zhou, Q. Z. Sheng, and L. Yao, "Dflstar: A decentralized federated learning framework with self-knowledge distillation and participant selection," in *International Conference on Information and Knowledge Management*, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:273497250`.

[31] I. Tyou, T. Murata, T. Fukami, Y. Takezawa, and K. Niwa, "A localized primal-dual method for centralized/decentralized federated learning robust to data heterogeneity," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 10, pp. 94–107, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:266577741`.

[32] C.-Y. Huang, K. Srinivas, X. Zhang, and X. Li, "Overcoming data and model heterogeneities in decentralized federated learning via synthetic anchors," *ArXiv*, vol. abs/2405.11525, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:269921935`.

[33] A. Sadiev, E. Borodich, A. Beznosikov, *et al.*, "Decentralized personalized federated learning: Lower bounds and optimal algorithm for all personalization modes," *EURO J. Comput. Optim.*, vol. 10, p. 100 041, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:246608311`.

[34] J. Beilharz, B. Pfitzner, R. Schmid, P. Geppert, B. Arnrich, and A. Polze, "Implicit model specialization through dag-based decentralized federated learning," *Proceedings of the 22nd International Middleware Conference*, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:240419693`.

[35] R. Dai, L. Shen, F. He, X. Tian, and D. Tao, "Dispfl: Towards communication-efficient personalized federated learning via decentralized sparse training," in *International Conference on Machine Learning*, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:249240036`.

[36] J. Cao, Z. Lian, W. Liu, Z. Zhu, and C. Ji, "Hadfl: Heterogeneity-aware decentralized federated learning framework," *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:243918943`.

[37] J. Liu, T. Che, Y. Zhou, *et al.*, "Aedfl: Efficient asynchronous decentralized federated learning with heterogeneous devices," *ArXiv*, vol. abs/2312.10935, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:266359500`.

[38] Y. Liao, Y. Xu, H.-Z. Xu, M. Chen, L. Wang, and C. Qiao, "Asynchronous decentralized federated learning for heterogeneous devices," *IEEE/ACM Transactions on Networking*, vol. 32, pp. 4535–4550, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:271236198`.

[39] S. Zhou, K. Xu, and G. Y. Li, "Communication-efficient decentralized federated learning via one-bit compressive sensing," *2024 IEEE 99th Vehicular Technology Conference (VTC2024-Spring)*, pp. 1–5, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:261395891`.

[40] H. Ye, L. Liang, and G. Y. Li, "Decentralized federated learning with unreliable communications," *IEEE Journal of Selected Topics in Signal Processing*, vol. 16, pp. 487–500, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:236924373`.

[41] Y. Liao, Y. Xu, H. Xu, L. Wang, and C. Qian, "Adaptive configuration for heterogeneous participants in decentralized federated learning," *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pp. 1–10, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:254246346`.

[42] S. Baghersalimi, T. Teijeiro, A. Aminifar, and D. A. Alonso, "Decentralized federated learning for epileptic seizures detection in low-power wearable systems," *IEEE Transactions on Mobile Computing*, vol. 23, pp. 6392–6407, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:263282600`.

[43] B. C. Tedeschini, S. Savazzi, R. Stoklasa, *et al.*, "Decentralized federated learning for healthcare networks: A case study on tumor segmentation," *IEEE Access*, vol. 10, pp. 8693–8708, 2022.

[44] Z. Lian, Q. Yang, W. Wang, *et al.*, "Deep-fel: Decentralized, efficient and privacy-enhanced federated edge learning for healthcare cyber physical systems," *IEEE Transactions on Network Science and Engineering*, vol. 9, pp. 3558–3569, 2022.

[45] M. Wei, W. Yu, and D. Chen, "Accdfl: Accelerated decentralized federated learning for healthcare iot networks," *IEEE Internet of Things Journal*, vol. 12, pp. 5329–5345, 2025.

[46] B. Li, W. Gao, J. Xie, M. Gong, L. Wang, and H. Li, "Prototype-based decentralized federated learning for the heterogeneous time-varying iot systems," *IEEE Internet of Things Journal*, vol. 11, pp. 6916–6927, 2024.

[47] Z. Lian and C. Su, "Decentralized federated learning for internet of things anomaly detection," *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022.

[48] T. Ma, H. Wang, and C. Li, "Quantized distributed federated learning for industrial internet of things," *IEEE Internet of Things Journal*, vol. 10, pp. 3027–3036, 2023.

[49] H. Ochiai, Y. Sun, Q. Jin, N. Wongwiwatchai, and H. Esaki, "Wireless ad hoc federated learning: A fully distributed cooperative machine learning," *ArXiv*, vol. abs/2205.11779, 2022.

[50] P. Pinyoanuntapong, W. H. Huff, M. Lee, C. Chen, and P. Wang, "Toward scalable and robust aiot via decentralized federated learning," *IEEE Internet of Things Magazine*, vol. 5, pp. 30–35, 2022.

[51] W. Qiu, W. Ai, H. Chen, Q. Feng, and G. Tang, "Decentralized federated learning for industrial iot with deep echo state networks," *IEEE Transactions on Industrial Informatics*, vol. 19, pp. 5849–5857, 2023.

[52] M. Elmahallawy, T. Luo, and M. I. Ibrahem, "Secure and efficient federated learning in leo constellations using decentralized key generation and on-orbit model aggregation," *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, pp. 5727–5732, 2023.

[53] C. Wu, Y. Zhu, and F. Wang, "Dsfl: Decentralized satellite federated learning for energy-aware leo constellation computing," *2022 IEEE International Conference on Satellite Computing (Satellite)*, pp. 25–30, 2022.

[54] Z.-H. Yan and D. Li, "Convergence time optimization for decentralized federated learning with leo satellites via number control," *IEEE Transactions on Vehicular Technology*, vol. 73, pp. 4517–4522, 2024.

[55] M. Yang, J. Zhang, and S. Liu, "Dfedsat: Communication-efficient and robust decentralized federated learning for leo satellite constellations," *ArXiv*, vol. abs/2407.05850, 2024.

[56] Z. Zhai, Q. Wu, S. Yu, R. Li, F. Zhang, and X. Chen, "Fedleo: An offloading-assisted decentralized federated learning framework for low earth orbit satellite networks," *IEEE Transactions on Mobile Computing*, vol. 23, pp. 5260–5279, 2024.

[57] L. Barbieri, M. Brambilla, and M. Nicoli, "A compressed decentralized federated learning frame work for enhanced environmental awareness in v2v networks," *2024 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*, pp. 374–378, 2024.

[58] E. T. M. Beltrán, P. M. S. Sánchez, G. Bovet, B. Stiller, G. M. Pérez, and A. H. Celdrán, "Flighter: Decentralized federated learning and situational awareness for secure military aerial reconnaissance," *IEEE Communications Magazine*, vol. 63, pp. 136–142, 2025.

[59] H. He, J. Du, C. Jiang, J. Wang, and J. Song, "Mobility-aware decentralized federated learning for autonomous underwater vehicles," *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, pp. 2347–2352, 2024.

[60] D. Pan, M. A. Khoshkholghi, and T. Mahmoodi, "Decentralized federated learning methods for reducing communication cost and energy consumption in uav networks," in *Mobile Computing, Applications, and Services*, 2023.

[61] Y. Qu, H. Dai, Z. Yan, *et al.*, "Decentralized federated learning for uav networks: Architecture, challenges, and opportunities," *IEEE Network*, vol. 35, pp. 156–162, 2021.

[62] G. Tan, H. Yuan, H. Hu, S. Zhou, and Z. Zhang, "A framework of decentralized federated learning with soft clustering and 1-bit compressed sensing for vehicular networks," *IEEE Internet of Things Journal*, vol. 11, pp. 23 617–23 629, 2024.

[63] Y. Xiao, Y. Ye, S. Huang, *et al.*, "Fully decentralized federated learning-based on-board mission for uav swarm system," *IEEE Communications Letters*, vol. 25, pp. 3296–3300, 2021.

[64] K. Xiong, R. Wang, S. Leng, C. Huang, and C. Yuen, "Ris-empowered topology control for decentralized federated learning in urban air mobility," *IEEE Internet of Things Journal*, vol. 11, pp. 40 757–40 770, 2024.

[65] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[66] *TensorFlow Datasets, a collection of ready-to-use datasets*, `https://www.tensorflow.org/datasets`.

[67] M. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A survey of data partitioning and sampling methods to support big data analysis," *Big Data Min. Anal.*, vol. 3, pp. 85–101, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:218539779`.

[68] Q. Li, Y. Diao, Q. Chen, and B. He, "Federated learning on non-iid data silos: An experimental study," *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 965–978, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:231786564`.

[69] *Keras: Deep learning for humans*, `https://keras.io/`, Accessed: 2025-08-08.

[70] *Pytorch*, `https://pytorch.org/`, Accessed: 2025-08-10.

[71] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: `10.1038/s41586-020-2649-2`. [Online]. Available: `https://doi.org/10.1038/s41586-020-2649-2`.

[72] *Sparse - sparse*, `https://sparse.pydata.org/en/stable/`, Accessed: 2025-08-11.

[73] *Pickle — python object serialization — python 3.13.6 documentation*, `https://docs.python.org/3/library/pickle.html`, Accessed: 2025-08-13.

[74] S. Savazzi, M. Nicoli, and V. Rampa, "Federated learning with cooperating devices: A consensus approach for massive iot networks," *IEEE Internet of Things Journal*, vol. 7, pp. 4641–4654, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:209515403`.

[75] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor, "Tackling the objective inconsistency problem in heterogeneous federated optimization," *ArXiv*, vol. abs/2007.07481, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:220525591`.

[76] *Grpc*, `https://grpc.io/`, Accessed: 2025-04-13.

[77] *Protocol buffers documentation*, `https://protobuf.dev/`, Accessed: 2025-04-13.

[78] J. Konecný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *ArXiv*, vol. abs/1610.05492, 2016. [Online]. Available: `https://api.semanticscholar.org/CorpusID:14999259`.

[79] E. T. M. Beltrán, M. Q. Pérez, P. M. S. S'anchez, *et al.*, "Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, pp. 2983–3013, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:253523208`.

[80] J. Wu, F. Dong, H. Leung, Z. Zhu, J. Zhou, and S. Drew, "Topology-aware federated learning in edge computing: A comprehensive survey," *ACM Computing Surveys*, vol. 56, pp. 1–41, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:256616242`.

[81] H. Chen, H. Wang, Q. Long, D. Jin, and Y. Li, "Advancements in federated learning: Models, methods, and privacy," *ACM Computing Surveys*, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:257078992`.

[82] Q. W. Khan, A. Khan, A. Rizwan, R. Ahmad, S. Khan, and D.-H. Kim, "Decentralized machine learning training: A survey on synchronization, consolidation, and topologies," *IEEE Access*, vol. 11, pp. 68 031–68 050, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:259815892`.

[83] L. Yuan, L. Sun, P. S. Yu, and Z. Wang, "Decentralized federated learning: A survey and perspective," *IEEE Internet of Things Journal*, vol. 11, pp. 34 617–34 638, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:259064130`.

[84] E. Gabrielli, G. Pica, and G. Tolomei, "A survey on decentralized federated learning," *ArXiv*, vol. abs/2308.04604, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:260735643`.

[85] S. K. S. Thabet, B. Soltani, Y. Zhou, Q. Z. Sheng, and S. Wen, "Towards efficient decentralized federated learning: A survey," in *International Conference on Advanced Data Mining and Applications*, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:275280274`.

[86] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, pp. 1–19, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:219878182`.

[87] A. Koloskova, S. U. Stich, and M. Jaggi, "Decentralized stochastic optimization and gossip algorithms with compressed communication," in *International Conference on Machine Learning*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:59553565.

[88] J. Wang, A. K. Sahu, G. Joshi, and S. Kar, "Matcha: A matching-based link scheduling strategy to speed up distributed optimization," *IEEE Transactions on Signal Processing*, vol. 70, pp. 5208–5221, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253461410.

[89] Z. Tang, S. Shi, B. Li, and X. Chu, "Gossipfl: A decentralized federated learning framework with sparsified and adaptive communication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, pp. 909–922, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:255046867.

[90] H. Wang, L. Muñoz-González, M. Z. Hameed, D. Eklund, and S. Raza, "Sparsfa: Towards robust and communication-efficient peer-to-peer federated learning," *Comput. Secur.*, vol. 129, p. 103 182, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257566786.

[91] W. Liu, L. Chen, Y. Chen, and W. Wang, "Communication-efficient design for quantized decentralized federated learning," *IEEE Transactions on Signal Processing*, vol. 72, pp. 1175–1188, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257532612.

[92] L. Wang, Y. Xu, H. Xu, M. Chen, and L. Huang, "Accelerating decentralized federated learning in heterogeneous edge computing," *IEEE Transactions on Mobile Computing*, vol. 22, pp. 5001–5016, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:249144039.

[93] R. Zong, Y. Qin, F. Wu, Z. Tang, and K. Li, "Fedcs: Efficient communication scheduling in decentralized federated learning," *Inf. Fusion*, vol. 102, p. 102 028, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:262156336.

[94] V. Gupta, A. Luqman, N. Chattopadhyay, A. Chattopadhyay, and D. T. Niyato, "Travellingfl: Communication efficient peer-to-peer federated learning," *IEEE Transactions on Vehicular Technology*, vol. 73, pp. 5005–5019, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:265225957.

[95] A. Nedić and A. Olshevsky, "Distributed optimization over time-varying directed graphs," *52nd IEEE Conference on Decision and Control*, pp. 6855–6860, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:8361755.

[96] C. Lanza, T. Tuytelaars, M. Miozzo, E. Angelats, and P. Dini, "Joint class and domain continual learning for decentralized federated processes," *IEEE Access*, vol. 13, pp. 56 982–56 993, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277448727.

[97] D. Gabay and B. Mercier, "A dual algorithm for the solution of nonlinear variational problems via finite element approximation," *Computers & Mathematics With Applications*, vol. 2, pp. 17–40, 1976. [Online]. Available: https://api.semanticscholar.org/CorpusID:174707.

[98] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, pp. 1–122, 2011. [Online]. Available: `https://api.semanticscholar.org/CorpusID:51789432`.

[99] W. Shi, Q. Ling, K. Yuan, G. Wu, and W. Yin, "On the linear convergence of the admm in decentralized consensus optimization," *IEEE Transactions on Signal Processing*, vol. 62, pp. 1750–1761, 2013. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5642927`.

[100] W. W. Hager and H. Zhang, "Inexact alternating direction methods of multipliers for separable convex optimization," *Computational Optimization and Applications*, vol. 73, pp. 201–235, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:37330523`.

[101] C. Bucila, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Knowledge Discovery and Data Mining*, 2006. [Online]. Available: `https://api.semanticscholar.org/CorpusID:11253972`.

[102] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *ArXiv*, vol. abs/1503.02531, 2015. [Online]. Available: `https://api.semanticscholar.org/CorpusID:7200347`.

[103] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4320–4328, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:26071966`.

[104] P. T. Boufounos and R. Baraniuk, "1-bit compressive sensing," *2008 42nd Annual Conference on Information Sciences and Systems*, pp. 16–21, 2008. [Online]. Available: `https://api.semanticscholar.org/CorpusID:206563812`.

[105] P. V. Dantas, W. S. da Silva, L. C. Cordeiro, and C. B. Carvalho, "A comprehensive review of model compression techniques in machine learning," *Appl. Intell.*, vol. 54, pp. 11 804–11 844, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:272362488`.

[106] B. Hasircioglu and D. Gunduz, "Communication efficient private federated learning using dithering," *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7575–7579, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:261822187`.

[107] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," *ArXiv*, vol. abs/1704.05021, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:2140766`.

[108] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," *ArXiv*, vol. abs/1809.07599, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:52307874`.

[109] Q. Li, Z. Wen, Z. Wu, and B. He, "A survey on federated learning systems: Vision, hype and reality for data privacy and protection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, pp. 3347–3366, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:198179889`.

[110] S. Wang, T. Tuor, T. Salonidis, *et al.*, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 1205–1221, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:51921962`.

[111] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur, "Improving deep neural network acoustic models using generalized maxout networks," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 215–219, 2014. [Online]. Available: `https://api.semanticscholar.org/CorpusID:1129207`.

[112] T. Lin, S. U. Stich, and M. Jaggi, "Don't use large mini-batches, use local sgd," *ArXiv*, vol. abs/1808.07217, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:52071640`.

[113] K. I. Tsianos, S. F. Lawlor, and M. G. Rabbat, "Communication/computation tradeoffs in consensus-based distributed optimization," in *Neural Information Processing Systems*, 2012. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5683083`.

[114] M. Kamp, L. Adilova, J. Sicking, *et al.*, "Efficient decentralized deep learning by dynamic model averaging," *ArXiv*, vol. abs/1807.03210, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:49655200`.

[115] J. Zhang, H. Tu, Y. Ren, *et al.*, "An adaptive synchronous parallel strategy for distributed machine learning," *IEEE Access*, vol. 6, pp. 19 222–19 230, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5039234`.

[116] C. Hu, J. Jiang, and Z. Wang, "Decentralized federated learning: A segmented gossip approach," *ArXiv*, vol. abs/1908.07782, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:201124492`.

[117] L. Xiao and S. P. Boyd, "Fast linear iterations for distributed averaging," *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, vol. 5, 4997–5002 Vol.5, 2003. [Online]. Available: `https://api.semanticscholar.org/CorpusID:6001203`.

[118] Z. Tang, S. Shi, and X. Chu, "Communication-efficient decentralized learning with sparsification and adaptive peer selection," *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1207–1208, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:211259518`.

[119] F. Faghri, I. Tabrizian, I. Markov, D. Alistarh, D. M. Roy, and A. Ramezani-Kebrya, "Adaptive gradient quantization for data-parallel sgd," *ArXiv*, vol. abs/2010.12460, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:225062069`.

[120] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *Neural Information Processing Systems*, 2016. [Online]. Available: `https://api.semanticscholar.org/CorpusID:263894534`.

[121] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith, "On the convergence of federated optimization in heterogeneous networks," *ArXiv*, vol. abs/1812.06127, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:56321517`.

[122] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," in *Neural Information Processing Systems*, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:1467846`.

[123] J. Wang and G. Joshi, "Cooperative sgd: A unified framework for the design and analysis of communication-efficient sgd algorithms," *ArXiv*, vol. abs/1808.07576, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:52078222`.

[124] T. Sun, D. Li, and B. Wang, "Decentralized federated averaging," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, pp. 4289–4301, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:233387998`.

[125] Y. Shi, L. Shen, K. Wei, *et al.*, "Improving the model consistency of decentralized federated learning," *ArXiv*, vol. abs/2302.04083, 2023. [Online]. Available: `https://api.semanticscholar.org/CorpusID:256662397`.

[126] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *arXiv: Learning*, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:59316566`.

[127] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. A. Jarvis, "Safa: A semi-asynchronous protocol for fast federated learning with low overhead," *IEEE Transactions on Computers*, vol. 70, pp. 655–668, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:203642110`.

[128] J.-G. Park, D.-J. Han, M. Choi, and J. Moon, "Sageflow: Robust federated learning against both stragglers and adversaries," in *Neural Information Processing Systems*, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:245019704`.

[129] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," *ArXiv*, vol. abs/1710.06952, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:22451897`.

[130] M. Chen, B. Mao, and T. Ma, "Fedsa: A staleness-aware asynchronous federated learning algorithm with non-iid data," *Future Gener. Comput. Syst.*, vol. 120, pp. 1–12, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:233374925`.

[131] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous online federated learning for edge devices with non-iid data," *2020 IEEE International Conference on Big Data (Big Data)*, pp. 15–24, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:224896087`.

[132] J. Nguyen, K. Malik, H. Zhan, *et al.*, "Federated learning with buffered asynchronous aggregation," *ArXiv*, vol. abs/2106.06639, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:235422293`.

[133] N. Su and B. Li, "How asynchronous can federated learning be?" *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pp. 1–11, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:248837811`.

[134] M. Lin, R. Ji, Y. Wang, *et al.*, "Hrank: Filter pruning using high-rank feature map," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1526–1535, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:211258761`.

[135] H. Zhang, J. Liu, J. Jia, Y. Zhou, H. Dai, and D. Dou, "Fedduap: Federated learning with dynamic update and adaptive pruning using shared data on the server," in *International Joint Conference on Artificial Intelligence*, 2022. [Online]. Available: `https://api.semanticscholar.org/CorpusID:248377019`.

[136] S. Yu, Z. Yao, A. Gholami, Z. Dong, M. W. Mahoney, and K. Keutzer, "Hessian-aware pruning and optimal neural implant," *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 3665–3676, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:231693233`.

[137] I. Hegedüs, G. Danner, and M. Jelasity, "Gossip learning as a decentralized alternative to federated learning," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:174800884`.

[138] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger, "Braintorrent: A peer-to-peer environment for decentralized federated learning," *ArXiv*, vol. abs/1905.06731, 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:155099936`.

[139] S. Savazzi, M. Nicoli, V. Rampa, and S. Kianoush, "Federated learning with mutually cooperating devices: A consensus approach towards server-less model optimization," *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3937–3941, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:216480499`.

[140] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, pp. 2278–2324, 1998. [Online]. Available: `https://api.semanticscholar.org/CorpusID:14542261`.

[141] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *ArXiv*, vol. abs/1708.07747, 2017. [Online]. Available: `https://api.semanticscholar.org/CorpusID:702279`.

[142] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, University of Toronto, 2009. [Online]. Available: `https://api.semanticscholar.org/CorpusID:18268744`.

[143] M. Assran and M. G. Rabbat, "Asynchronous gradient push," *IEEE Transactions on Automatic Control*, vol. 66, pp. 168–183, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:4384739`.

[144] M. Assran, N. Loizou, N. Ballas, and M. G. Rabbat, "Stochastic gradient push for distributed deep learning," *ArXiv*, vol. abs/1811.10792, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:53753741`.

[145] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu, "Communication-efficient decentralized machine learning over heterogeneous networks," *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 384–395, 2020. [Online]. Available: `https://api.semanticscholar.org/CorpusID:221655553`.

[146] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:195908774.

[147] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:14124313.

[148] G. Cohen, S. Afshar, J. C. Tapson, and A. van Schaik, "Emnist: Extending mnist to handwritten letters," *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2921–2926, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:30587588.

[149] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, pp. 211–252, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:2930547.

[150] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag, "Pascal large scale learning challenge," *25th International Conference on Machine Learning (ICML2008) Workshop. J. Mach. Learn. Res*, vol. 10, pp. 1937–1953, Jan. 2008.

[151] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:11027141.

[152] T. C. Aysal, M. J. Coates, and M. G. Rabbat, "Distributed average consensus with dithered quantization," *IEEE Transactions on Signal Processing*, vol. 56, pp. 4905–4918, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:1328785.

[153] R. Carli, F. Fagnani, P. Frasca, T. Taylor, and S. Zampieri, "Average consensus on networks with transmission noise or quantization," *2007 European Control Conference (ECC)*, pp. 1852–1857, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:1200312.

[154] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu, "Communication compression for decentralized training," in *Neural Information Processing Systems*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52891696.

[155] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:206594692.

[156] H. Wang, M. Yurochkin, Y. Sun, D. Papailiopoulos, and Y. Khazaeni, "Federated learning with matched averaging," *ArXiv*, vol. abs/2002.06440, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211132598.

[157] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, 2011, p. 7. [Online]. Available: https://api.semanticscholar.org/CorpusID:16852518.

[158] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *International Conference on Artificial Intelligence and Statistics*, 2011. [Online]. Available: `https://api.semanticscholar.org/CorpusID:308212`.

[159] A. Bellet, A.-M. Kermarrec, and E. Lavoie, "D-cliques: Compensating for data heterogeneity with topology in decentralized federated learning," *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pp. 1–11, 2021. [Online]. Available: `https://api.semanticscholar.org/CorpusID:243757009`.

[160] D. Weissteiner, *Gradient evolution — david weissteiner*, `https://ywcb00.ywcb.org/blog/2024/gradient-evolution/`, Accessed: 2025-10-23.

[161] T. Chen, G. B. Giannakis, T. Sun, and W. Yin, "Lag: Lazily aggregated gradient for communication-efficient distributed learning," in *Neural Information Processing Systems*, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:44061071`.

[162] J. Zhang, H. Tu, Y. Ren, *et al.*, "An adaptive synchronous parallel strategy for distributed machine learning," *IEEE Access*, vol. 6, pp. 19 222–19 230, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:5039234`.

[163] M. Kamp, L. Adilova, J. Sicking, *et al.*, "Efficient decentralized deep learning by dynamic model averaging," *ArXiv*, vol. abs/1807.03210, 2018. [Online]. Available: `https://api.semanticscholar.org/CorpusID:49655200`.

[164] M. Theologitis, G. Frangias, G. Anestis, V. Samoladas, and A. Deligiannakis, "Communication-efficient distributed deep learning via federated dynamic averaging," in *International Conference on Extending Database Technology*, 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:270199927`.

[165] R. A. Fisher, *Iris*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C56C76, 1936.

[166] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[167] J. Browniee, *Multi-class classification tutorial with the keras deep learning library*, `https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/`, Accessed: 2025-11-11.

[168] D. Roussis, *Svhn classification with cnn (keras - ~96% acc)*, `https://www.kaggle.com/code/dimitriosroussis/svhn-classification-with-cnn-keras-96-acc`, Accessed: 2025-11-11.

[169] P. S. Krishna, *Salary_data.csv*, `https://gist.github.com/saikrishnapotluri/33ace369025ec4de0dfb9f22a0c5b09f`, Accessed: 2025-11-15.

A. Appendix

A.1. Learning Rate Grid Search

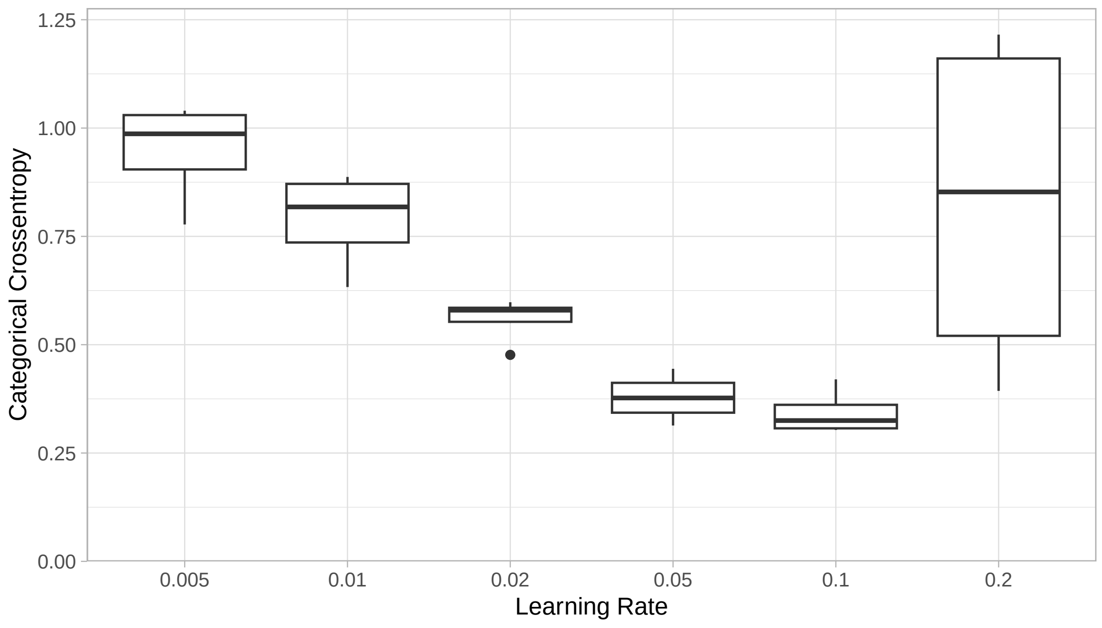


Figure A.1.: Box plot visualizing the final validation loss for seed values 13, 14, 15, and 16 over different learning rate values (x-axis), resulting from training on the Iris dataset in the experimental setting of Baseline 1 in Section 5.4.2. Learning rate 0.1 performs best.

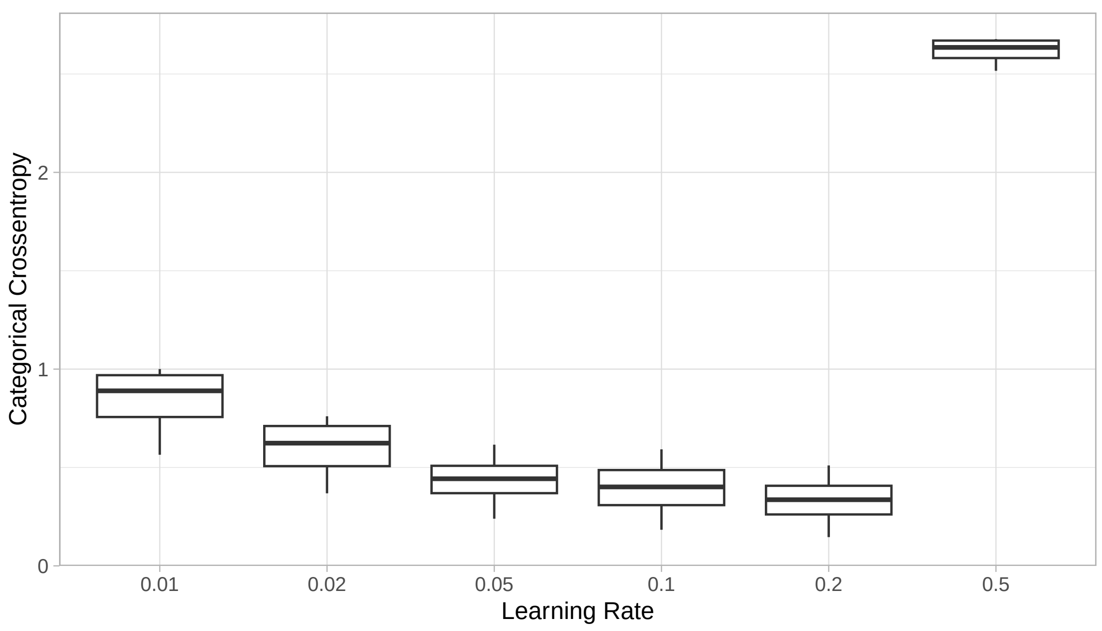


Figure A.2.: Box plot visualizing the final validation loss for seed values 13, 14, 15, and 16 over different learning rate values (x-axis), resulting from training on the Mnist dataset in the experimental setting of Baseline 1 in Section 5.4.2. Learning rate 0.2 performs best.

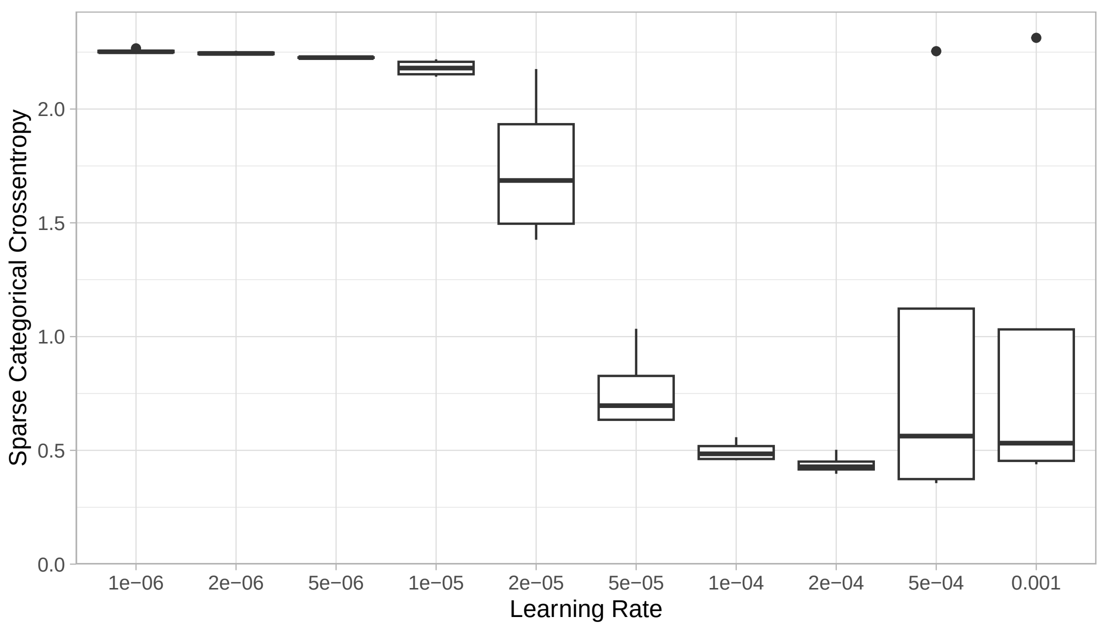


Figure A.3.: Box plot visualizing the final validation loss for seed values 13, 14, 15, and 16 over different learning rate values (x-axis), resulting from training on the SVHN dataset in the experimental setting of Baseline 1 in Section 5.4.2. Learning rate 0.0002 performs best.

A.2. Choice of Sensitivity Parameter

Experiment		SEED 666		SEED 667		SEED 668		SEED 669	
		Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
Baseline 1		0.4827	100	0.5441	100	0.5110	100	0.2893	100
Baseline 2		0.3913	51	0.6352	51	0.5380	51	0.2934	51
Baseline 3		0.3654	21	0.6212	21	0.5930	21	0.3458	21
Baseline 4		0.4108	11	0.7794	11	0.6974	11	0.4022	11
GT	$\theta_\rho = 1$	0.4653	98	0.6615	95	0.5090	96	0.2901	99
	$\theta_\rho = 2$	0.4656	92	0.6642	28	0.5093	90	0.2066	90
	$\theta_\rho = 3$	0.2886	82	0.7854	5	0.5538	78	0.4632	5
	$\theta_\rho = 4$	0.3076	42	0.8461	4	0.5129	73	0.5235	4
	$\theta_\rho = 5$	0.3059	46	0.8254	4	0.5684	61	0.5077	4

Table A.1.: Validation loss and number of synchronizations ('Sync.') of experiments on the Iris dataset for baselines and Gradient Thresholding (GT) with multiple values for parameter θ_ρ over different seeds. The gray shaded row marks our selected value for θ_ρ.

Experiment		SEED 666		SEED 667		SEED 668		SEED 669	
		Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
Baseline 1		0.2470	20	0.2774	20	0.2265	20	0.5016	20
Baseline 2		0.3999	11	0.2893	11	0.2296	11	0.4557	11
Baseline 3		2.4647	5	0.2899	5	0.2745	5	0.5348	5
Baseline 4		2.4486	4	0.3286	4	0.2894	4	0.5919	4
GT	$\theta_\rho = 1$	0.3237	20	0.2681	16	0.2274	16	0.4624	17
	$\theta_\rho = 1.25$	0.2491	18	0.3772	14	0.2308	13	0.4695	10
	$\theta_\rho = 1.5$	2.3760	17	0.2688	5	0.2332	9	0.5544	5
	$\theta_\rho = 1.75$	0.3748	8	0.3147	4	0.2617	5	0.5346	4
	$\theta_\rho = 2$	0.3847	11	0.2920	4	0.2942	4	0.5816	4
	$\theta_\rho = 2.25$	2.5542	7	0.3223	3	0.2937	4	0.5848	3
	$\theta_\rho = 2.5$	2.4151	6	0.2991	3	0.3092	4	0.6917	3
	$\theta_\rho = 2.75$	2.3923	6	0.3308	4	0.2974	4	0.7008	3
	$\theta_\rho = 3$	1.8258	8	0.2741	3	0.2797	3	0.6897	3

Table A.2.: Validation loss and number of synchronizations ('Sync.') of experiments on the Mnist dataset for baselines and Gradient Thresholding (GT) with multiple values for parameter θ_ρ over different seeds. The gray shaded row marks our selected value for θ_ρ.

Experiment		SEED 666		SEED 667		SEED 668		SEED 669	
		Loss	Sync.	Loss	Sync.	Loss	Sync.	Loss	Sync.
Baseline 1		0.3995	50	0.4202	50	0.3748	50	0.4186	50
Baseline 2		0.4215	26	0.3932	26	0.3769	26	0.3997	26
Baseline 3		0.4110	11	0.3721	11	0.4341	11	0.4433	11
Baseline 4		0.4701	6	0.4070	6	0.4729	6	0.5275	6
GT	$\theta_\rho = 1$	0.4046	48	0.4319	50	0.3787	48	0.4309	48
	$\theta_\rho = 2$	0.4088	31	0.3642	16	0.3832	23	0.3972	21
	$\theta_\rho = 3$	0.4300	9	0.3876	7	0.4237	9	0.4537	8
	$\theta_\rho = 4$	0.4458	8	0.3991	7	0.4190	8	0.4614	7
	$\theta_\rho = 5$	0.4698	7	0.4212	7	0.4209	7	0.4855	7

Table A.3.: Validation loss and number of synchronizations ('Sync.') of experiments on the SVHN dataset for baselines and Gradient Thresholding (GT) with multiple values for parameter θ_ρ over different seeds. The gray shaded row marks our selected value for θ_ρ.

A.3. Baseline Comparison

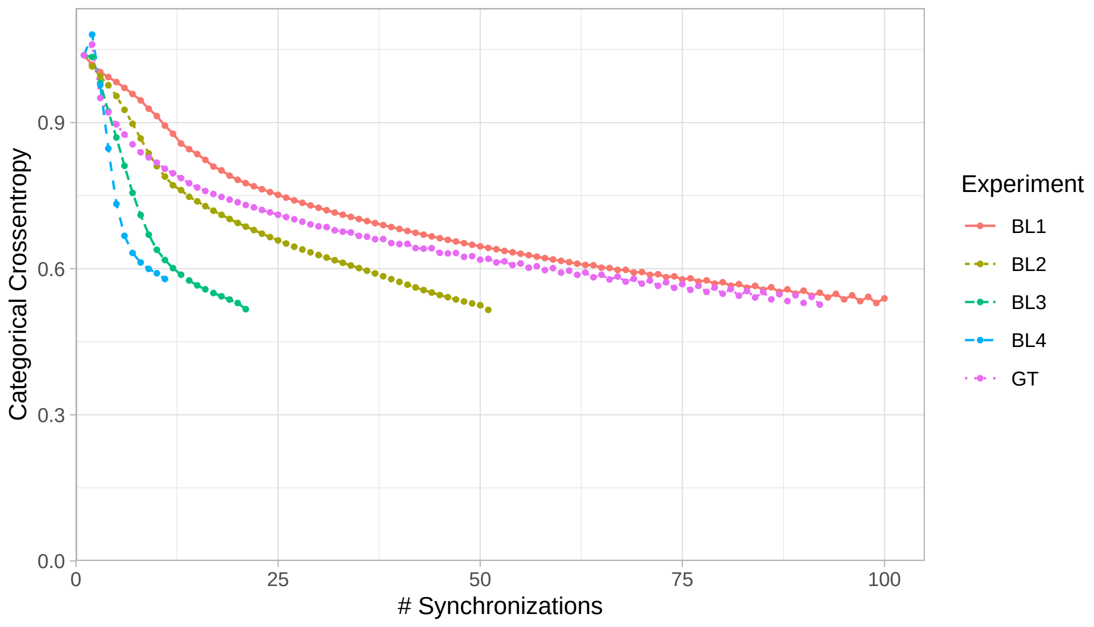


Figure A.4.: Test loss of experiments on the Iris dataset with seed 666 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

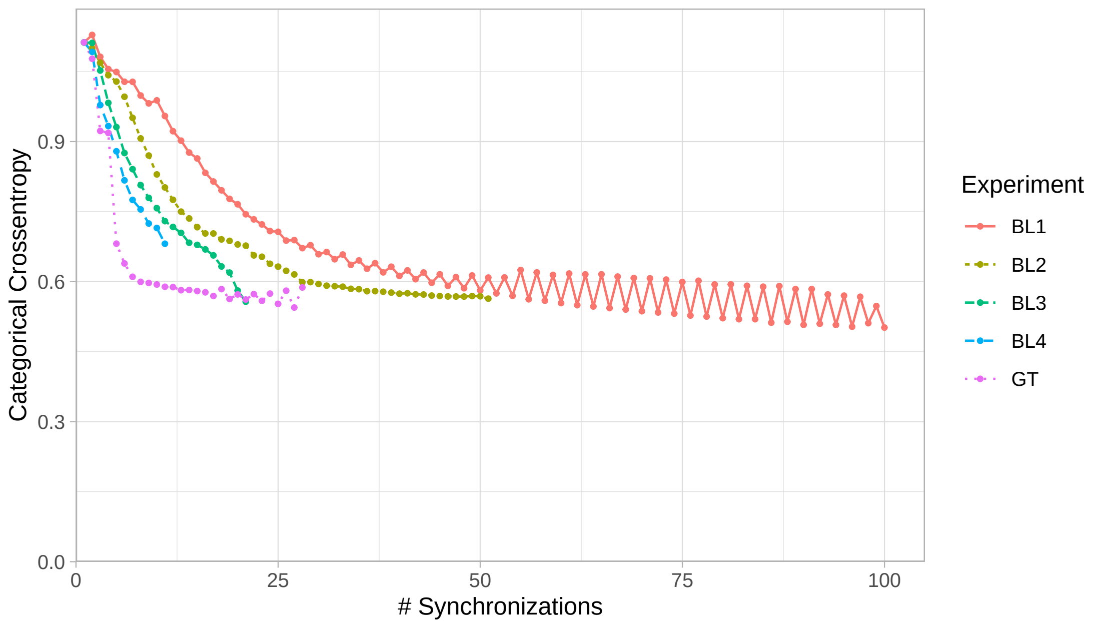


Figure A.5.: Test loss of experiments on the Iris dataset with seed 667 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

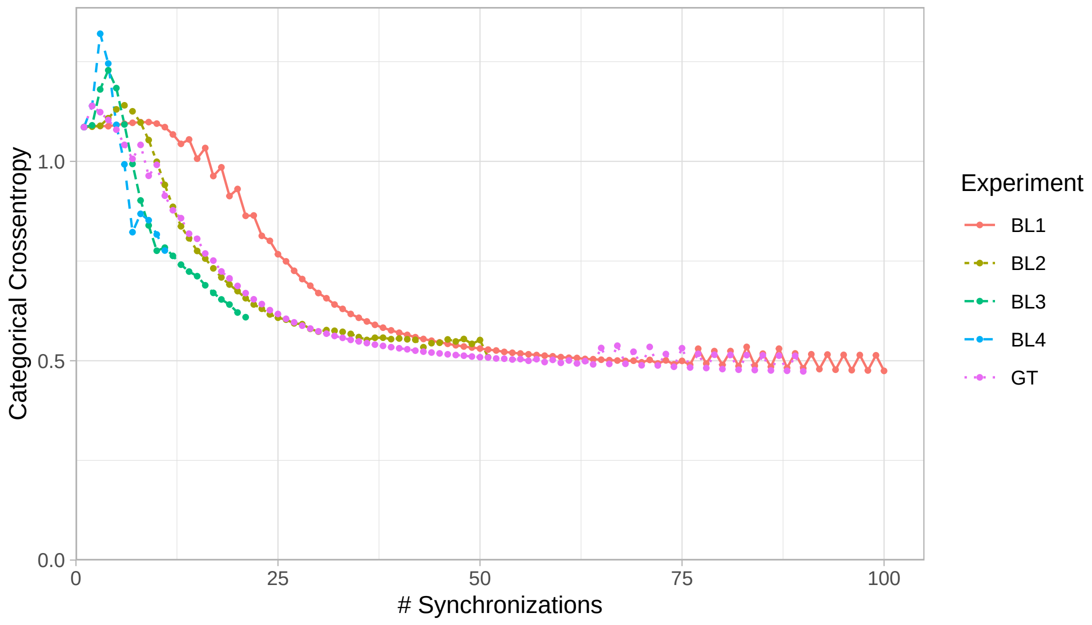


Figure A.6.: Test loss of experiments on the Iris dataset with seed 668 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

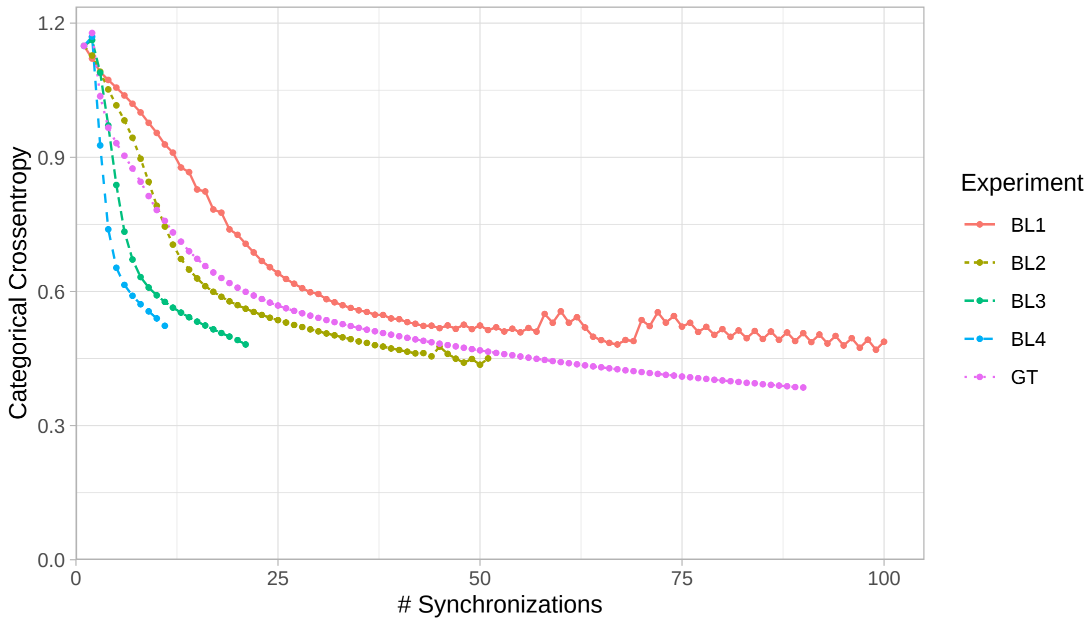


Figure A.7.: Test loss of experiments on the Iris dataset with seed 669 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

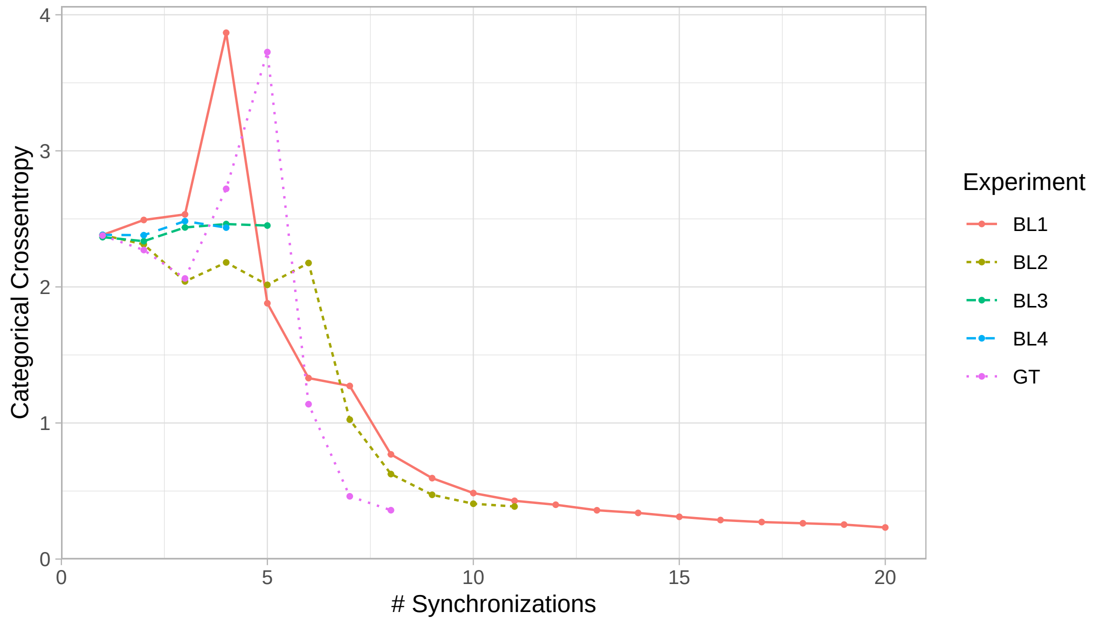


Figure A.8.: Test loss of experiments on the Mnist dataset with seed 666 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

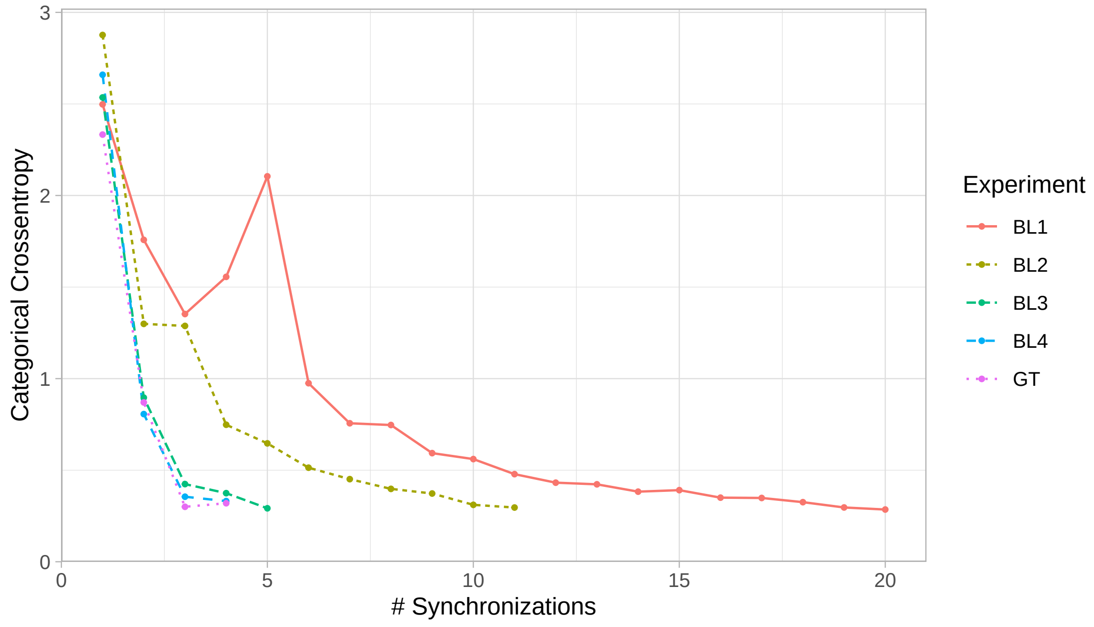


Figure A.9.: Test loss of experiments on the Mnist dataset with seed 667 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

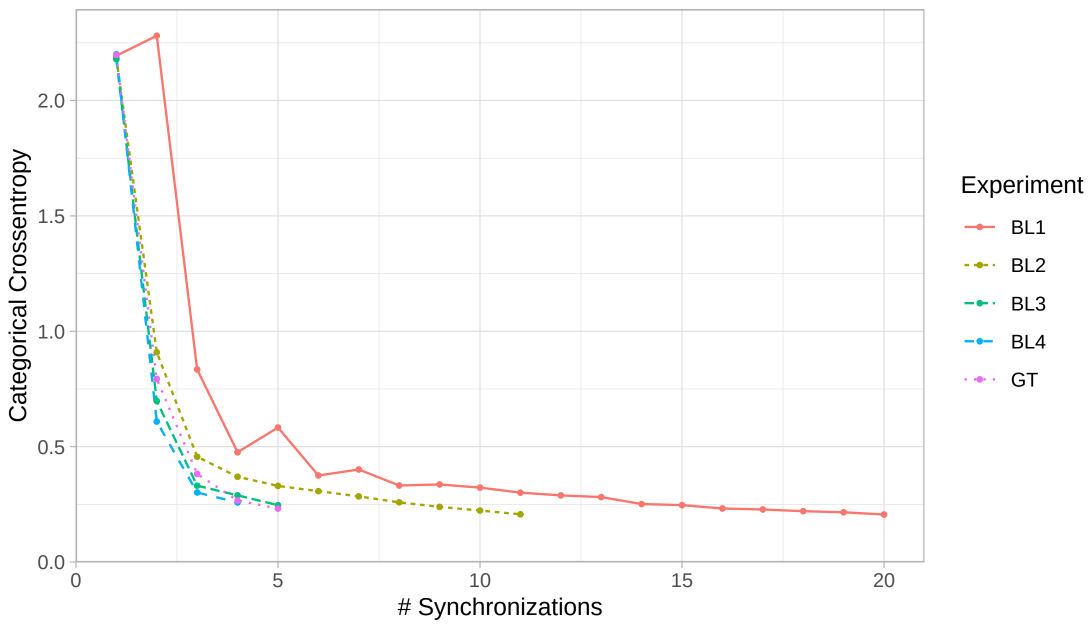


Figure A.10.: Test loss of experiments on the Mnist dataset with seed 668 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

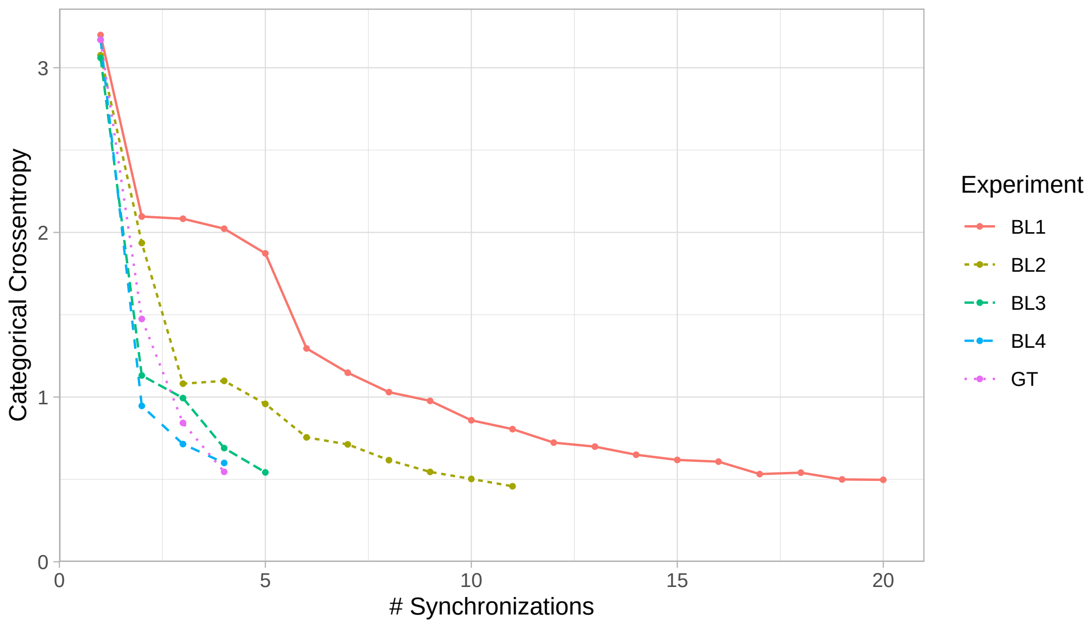


Figure A.11.: Test loss of experiments on the Mnist dataset with seed 669 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

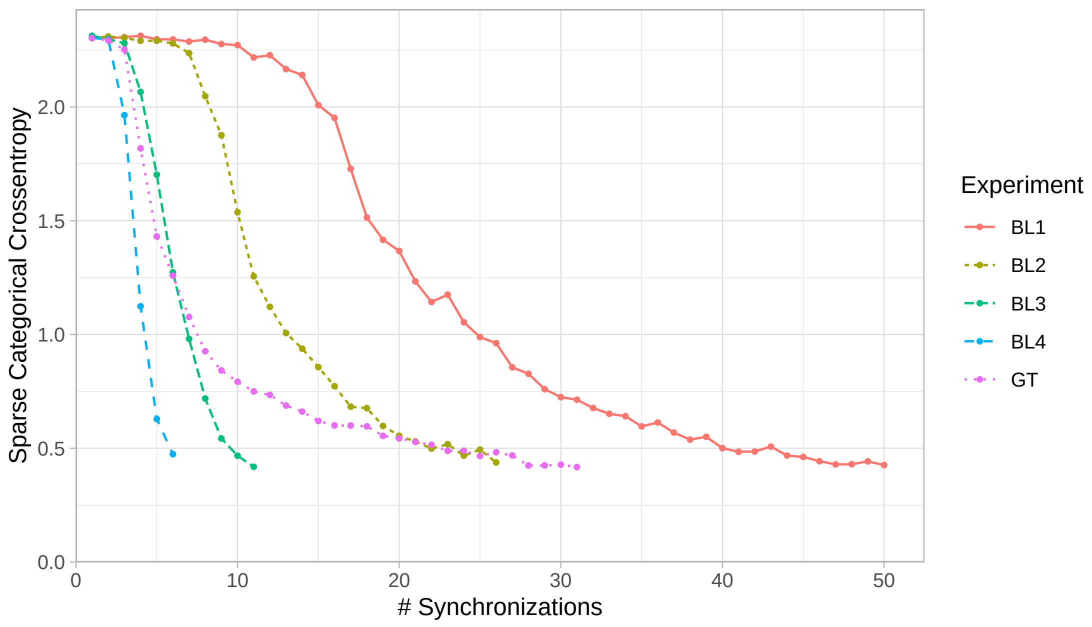


Figure A.12.: Test loss of experiments on the SVHN dataset with seed 666 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

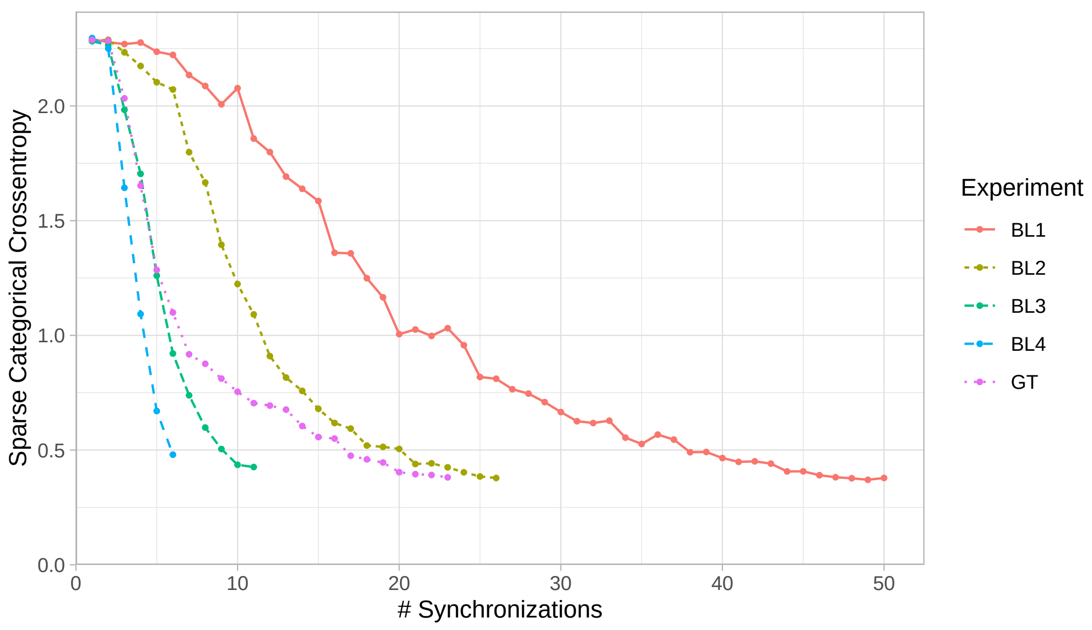


Figure A.13.: Test loss of experiments on the SVHN dataset with seed 668 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

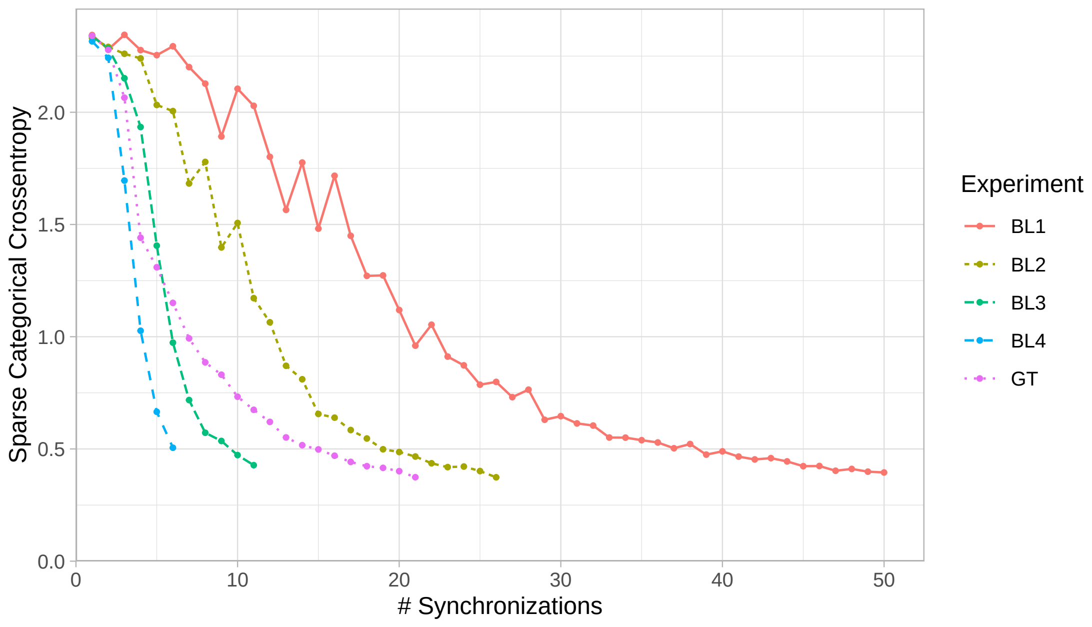


Figure A.14.: Test loss of experiments on the SVHN dataset with seed 669 over number of synchronizations for baselines (BL) and Gradient Thresholding (GT).

A.4. Threshold Distance

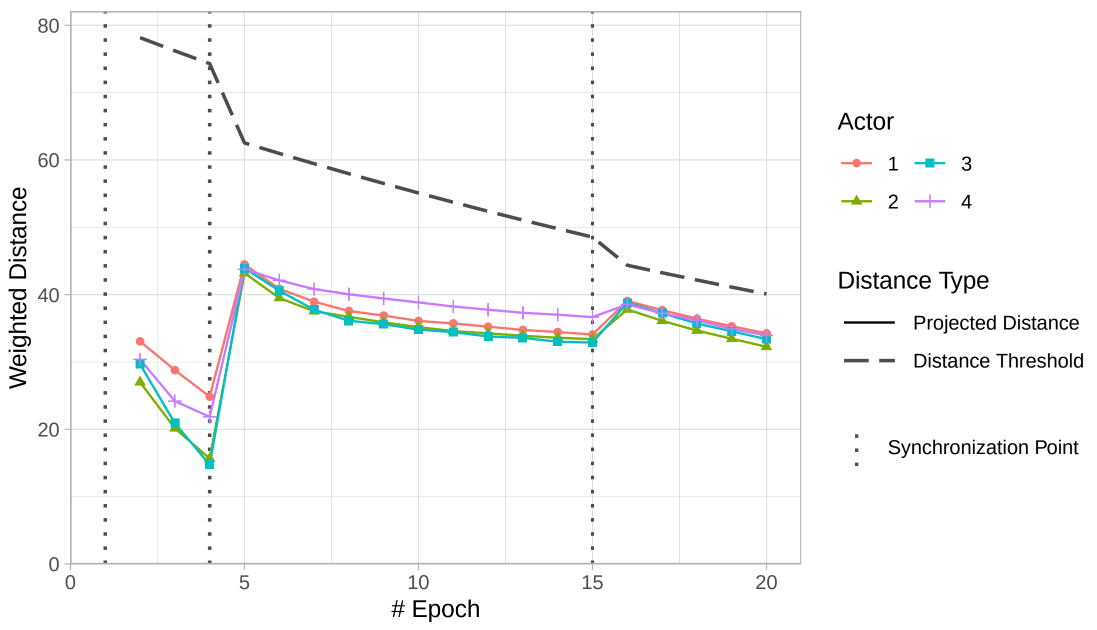


Figure A.15.: Projected distances of the individual actors in the experiment on the Mnist dataset with a seed value of 667. The dashed line represents the corresponding threshold boundary, which triggers a synchronization when exceeded. Synchronization points are indicated by dotted vertical lines.

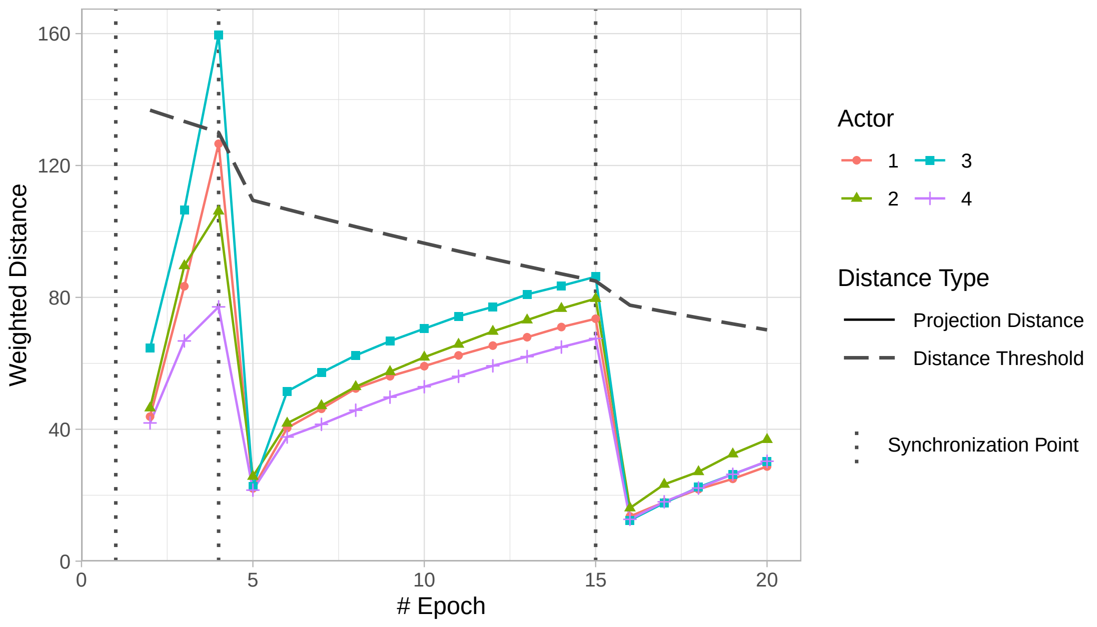


Figure A.16.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the Mnist dataset with a seed value of 667. The visual representation matches Figure A.15.

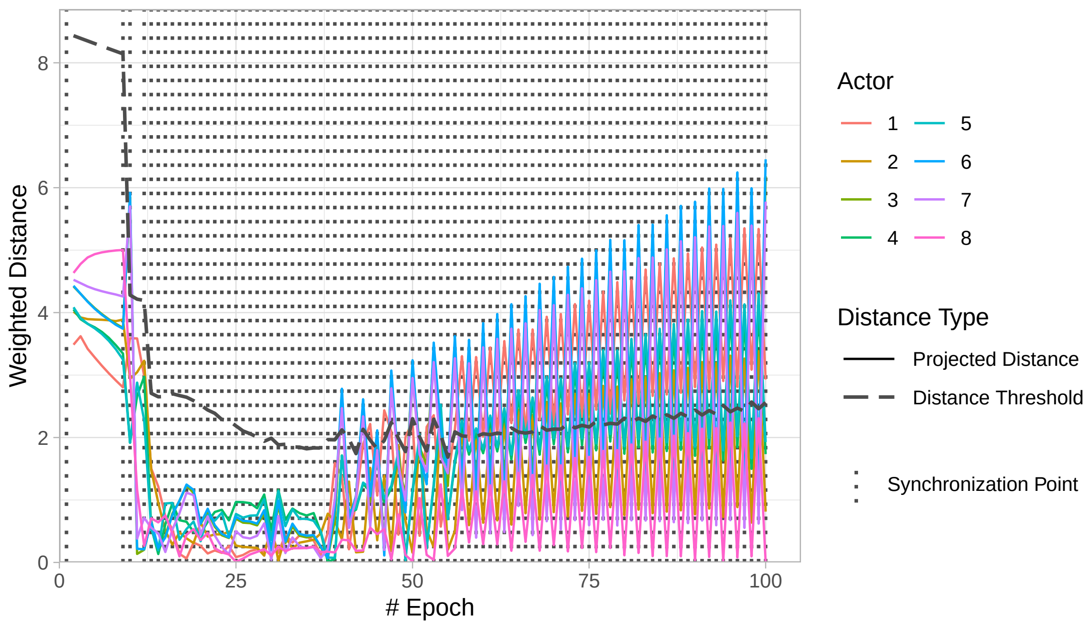


Figure A.17.: Projected distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the Iris dataset with a seed value of 666. The visual representation matches Figure A.15.

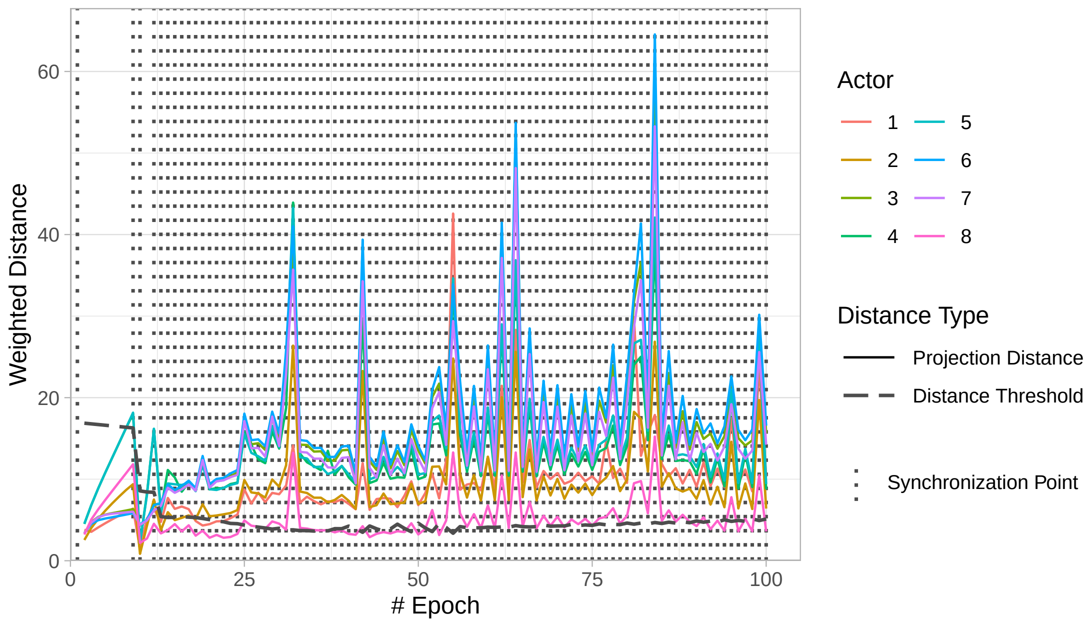


Figure A.18.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the Iris dataset with a seed value of 666. The visual representation matches Figure A.15.

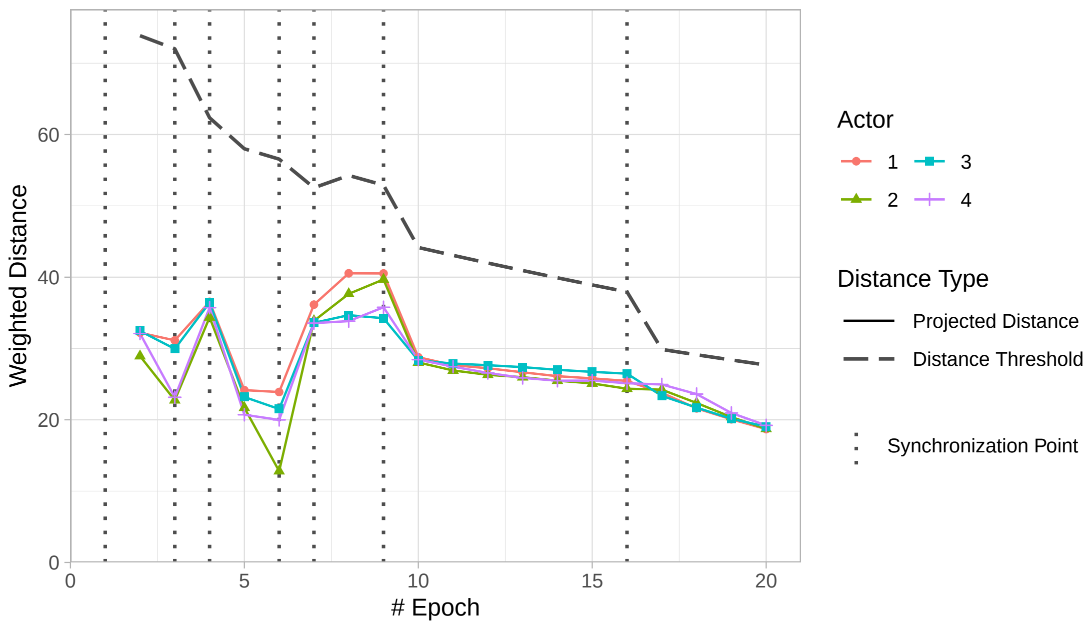


Figure A.19.: Projected distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the Mnist dataset with a seed value of 666. The visual representation matches Figure A.15.

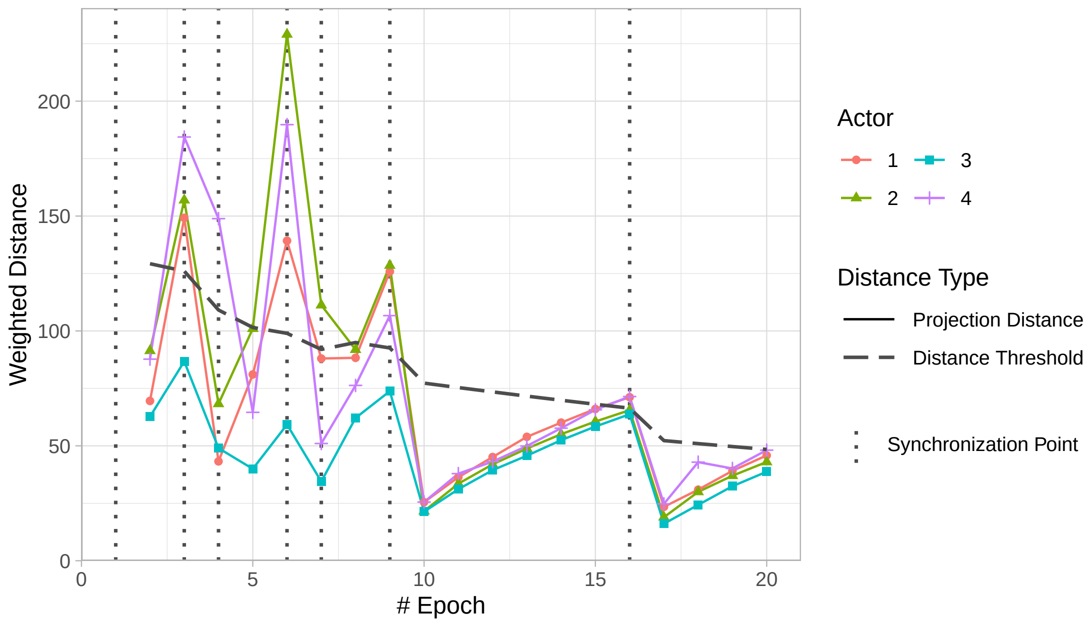


Figure A.20.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the Mnist dataset with a seed value of 666. The visual representation matches Figure A.15.

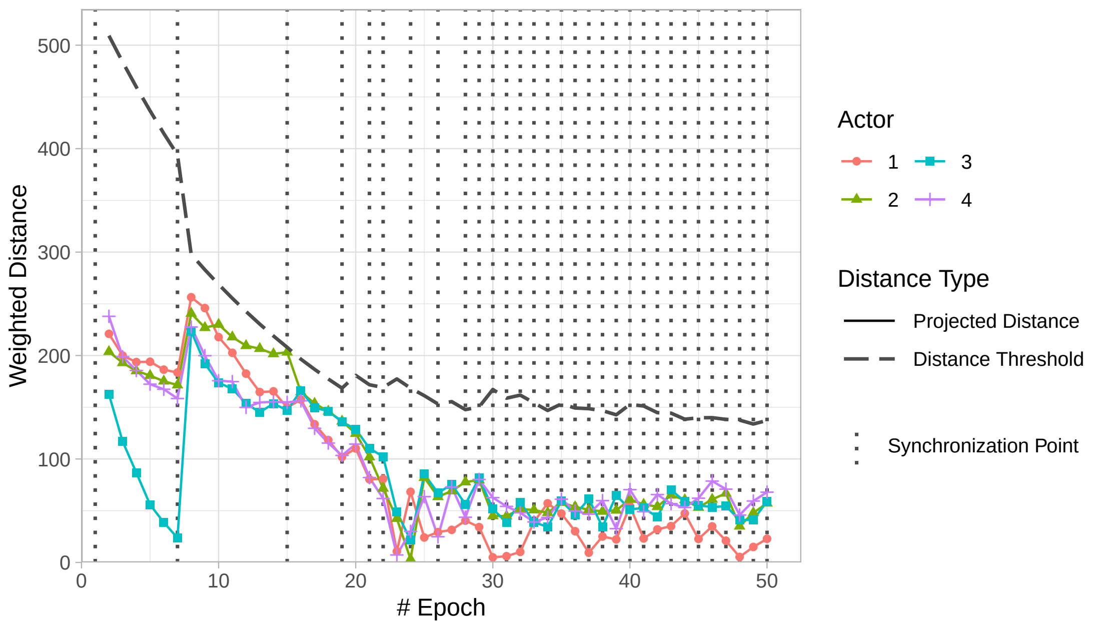


Figure A.21.: Projected distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the SVHN dataset with a seed value of 666. The visual representation matches Figure A.15.

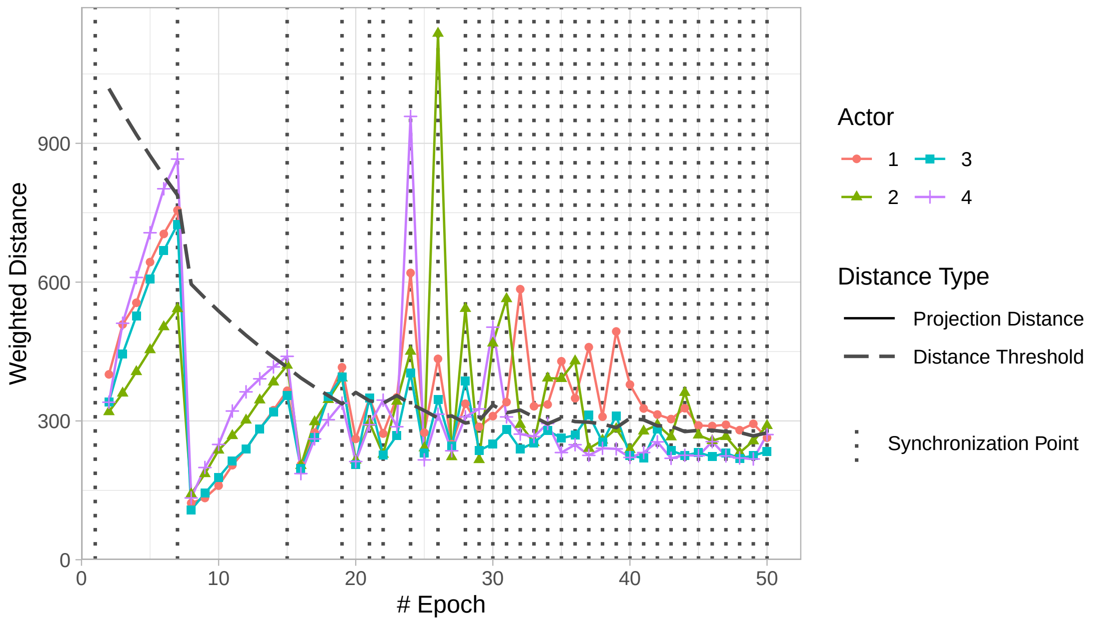


Figure A.22.: Projection distances of the individual actors with the corresponding threshold boundary (dashed line) in the experiment on the SVHN dataset with a seed value of 666. The visual representation matches Figure A.15.

A.5. Variability Test

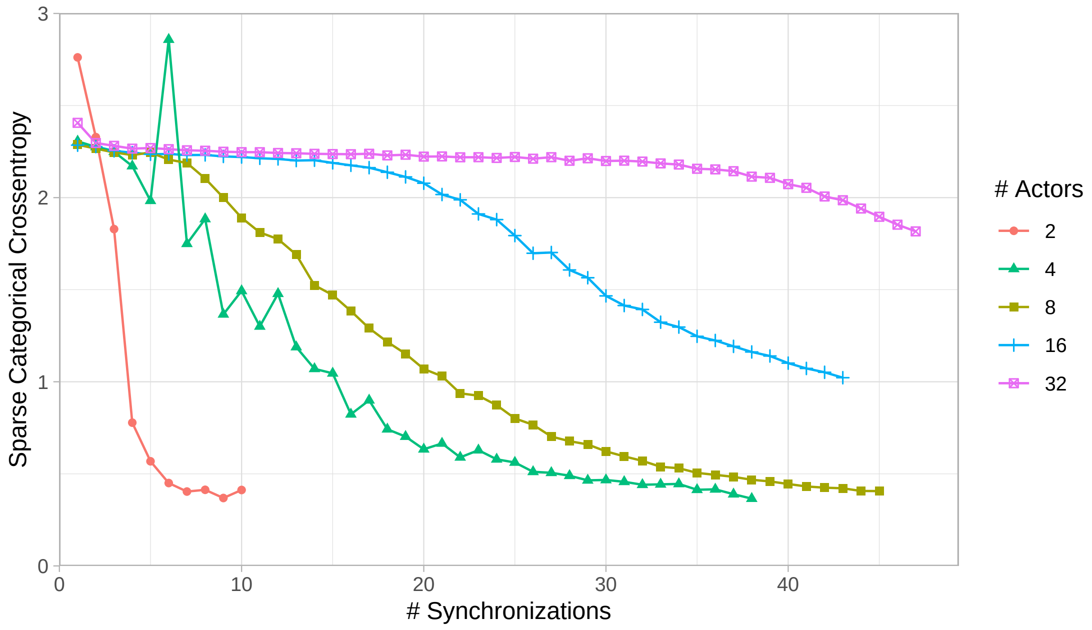


Figure A.23.: Test loss of experiments on the SVHN dataset with seed 14 for various numbers of actors over the number of synchronizations.

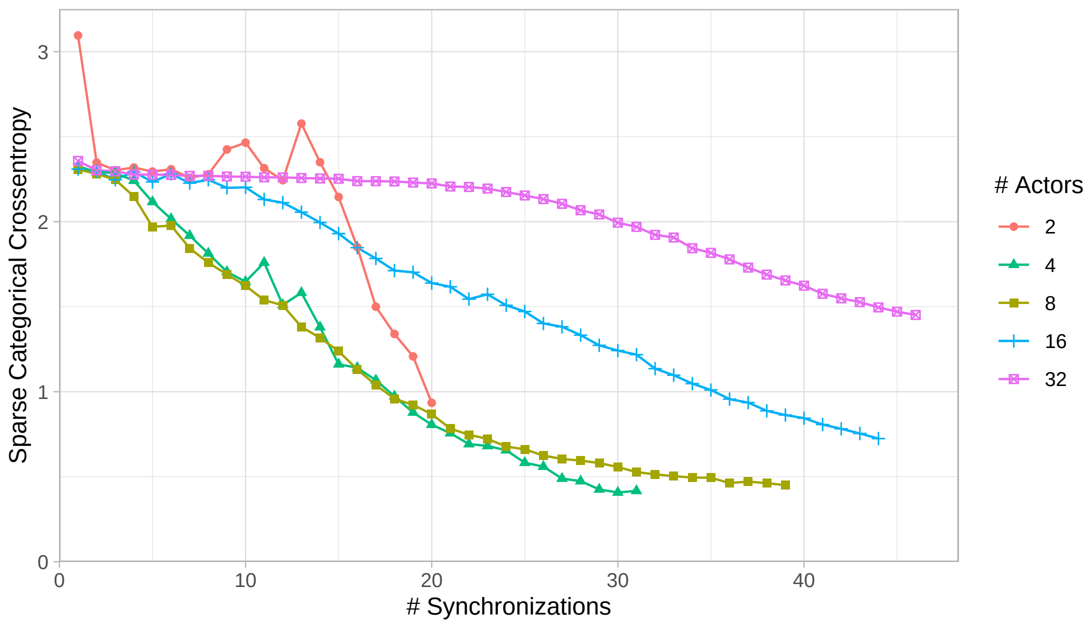


Figure A.24.: Test loss of experiments on the SVHN dataset with seed 15 for various numbers of actors over the number of synchronizations.

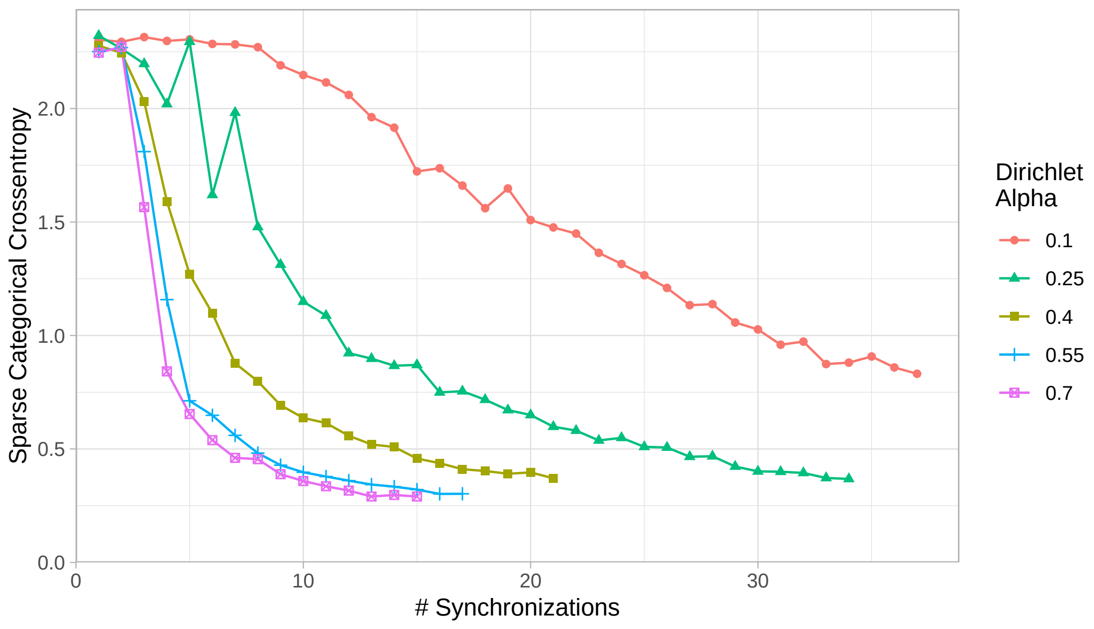


Figure A.25.: Test loss of experiments on the SVHN dataset with seed 14 for various values for the Dirichlet partitioning parameter α over the number of synchronizations.

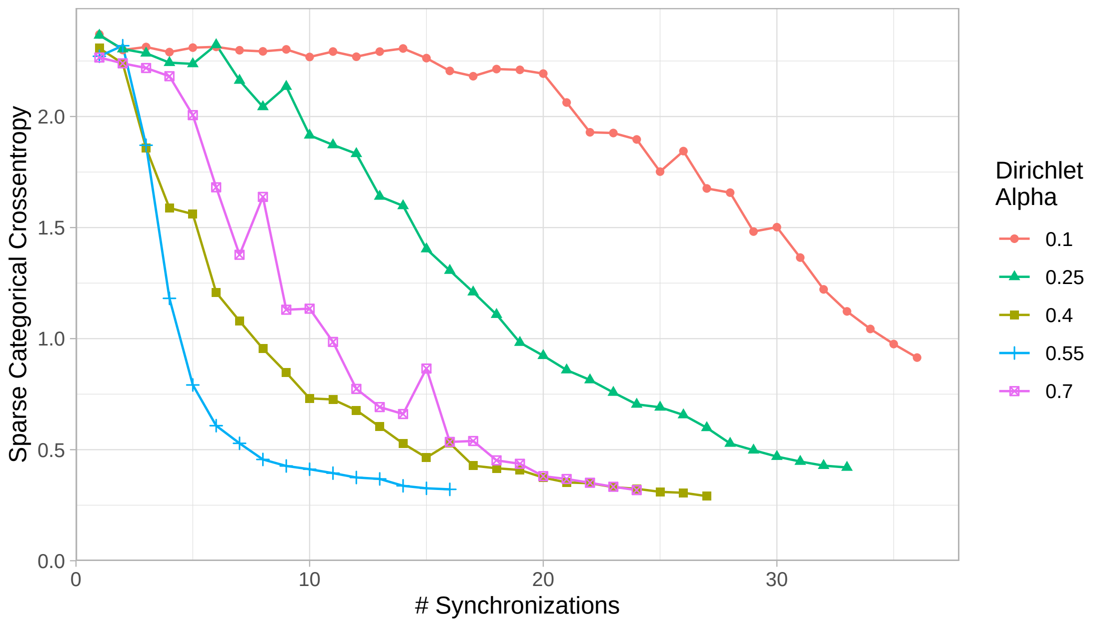


Figure A.26.: Test loss of experiments on the SVHN dataset with seed 15 for various values for the Dirichlet partitioning parameter α over the number of synchronizations.

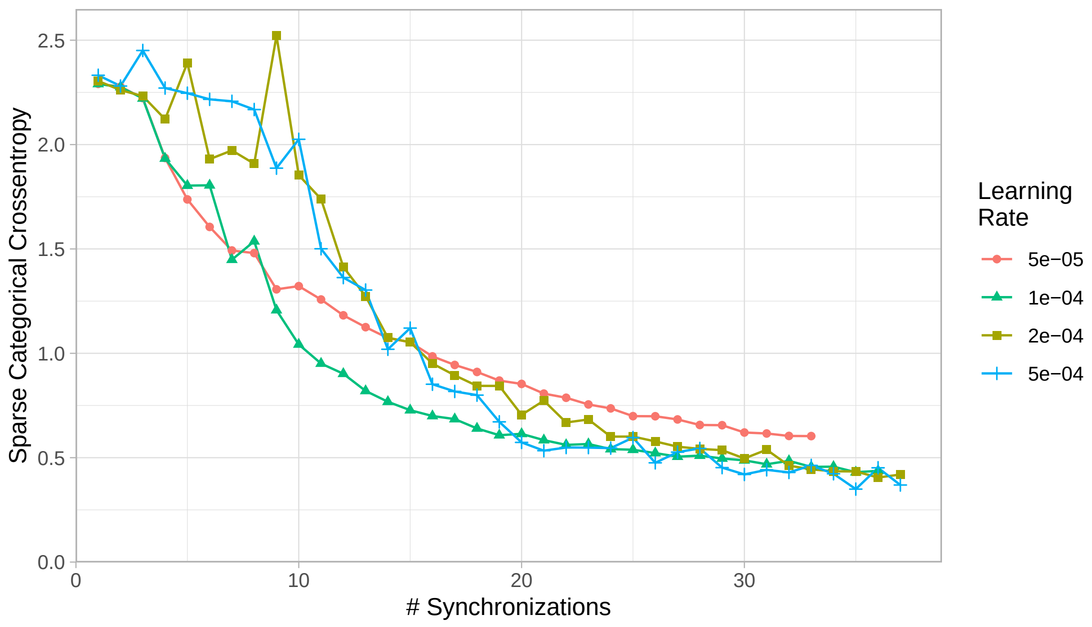


Figure A.27.: Test loss of experiments on the SVHN dataset with seed 14 for various learning rates over the number of synchronizations.

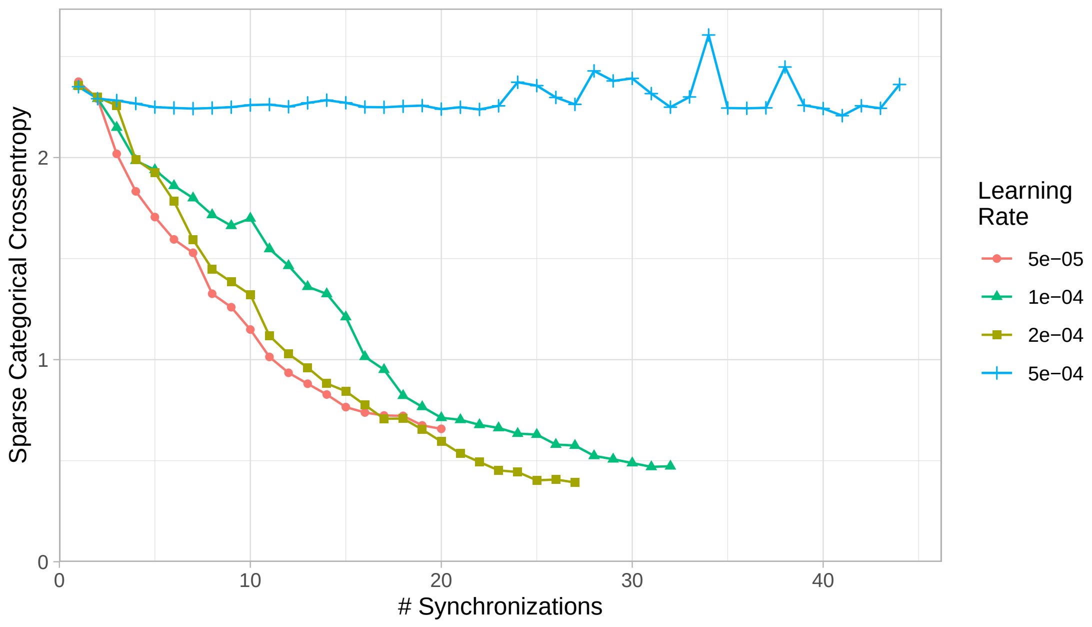


Figure A.28.: Test loss of experiments on the SVHN dataset with seed 15 for various learning rates over the number of synchronizations.