Computer Science
Master's Thesis

Improving communication-efficiency in Decentralized and Federated Learning Systems

Student:	Jeffrey Wigger
Address:	Kirchweg 2, 6033, Buchrain
Professor:	Prof. Dr. Anne-Marie Kermarrec
Assistants:	Rishi Sharma
	Dr. Rafael Pereira Pires

June 17, 2022

Scalable Computing Systems Laboratory
EPFL

Abstract

In recent years, privacy concerns regarding the sharing of personal data have not only become an essential concern for many individuals but also global policymakers. As a result, wide-ranging data protection laws have been passed that make the central storage of personal data increasingly impossible. Therefore, the need arose for new training methods for machine learning models that can be trained where the data is stored. decentralized learning (DL) attempts to solve precisely that problem. It gets by without needing a centralized entity by training the model where the data is stored and exchanging model parameters with other participants directly.

In this thesis, we aim to tackle one of decentralized learning's most significant challenges, the high cost of exchanging the model. We propose a new algorithm called JWINS++ (Just What Is Need Sharing) that adapts several compression methods for decentralized learning and combines them into a single algorithm. Gradient accumulation is adapted from centralized machine learning to act as a score that ranks parameters by their importance. The wavelet transformation from the field of electrical signal processing is applied to the model parameters and combined with the aforementioned model change accumulation to form an importance score in the wavelet domain. Finally, lossy floating point compression is applied to the wavelet frequency parameters chosen for sharing by the importance score.

The algorithm has been evaluated for several models and datasets belonging to different machine learning domains. For many models, JWINS++ achieves comparable accuracies to the full sharing baseline with a compression ratio of more than 20x and achieves even better accuracies for compression ratios of more than 8x. Additionally, it has been shown to work in asynchronous settings and dynamic topologies.

Contents

1 Introduction

Over the last decade, machine learning models have found a wide range of applications. For example, they have been successful in image recognition [26] and natural language processing task [9]. These models have now reached unprecedented scale, for example, GPT-3 [9] has several billion parameters and was trained on billions of samples. Hence, it has become unfeasible to train them on a single machine. Therefore, training is done using several machines, each with one or more GPUs or other hardware accelerators such as TPUs.

The field of distributed machine learning [3] has emerged to study the training of models in such settings. One of the fields' widely adapted techniques is the use of a parameter server (PS) [56] together with worker nodes. The former stores the current global model and updates it with gradients produced by the workers. Therefore, the computationally expensive process of computing the gradient can be distributed among many worker machines. These distribution benefits also extend to the dataset, as every worker may store only a fraction of the total data. An alternative to the synchronous parameter server architecture is using all-reduce to average the calculated gradients. The product of this all-reduce step is the same average gradient as on the parameter server. The only difference is that each worker will now use that gradient to update its model. The result of this operation corresponds to the same model that it would receive from the PS.

Both of these approaches come with their disadvantages. The parameter server's disadvantage is that it becomes a network bottleneck as it needs to receive all gradients and broadcast the new global model. The disadvantage of the all-reduce architecture is its inherent synchronous nature. Therefore, a single slow link or a straggling machine may slow down the entire training round.

Compressing the gradients can alleviate some of the communication load at the parameter server. Different compression schemes have been proposed, such as sending the TopK [19, 40, 54, 57] gradient values or quantizing [51, 1, 18] the gradients. However, these compression methods introduce errors that can negatively affect the convergence performance. Hence error correction methods like accumulation [51, 40] were introduced. The compression error is accumulated into an additional vector and reapplied to the gradients before they are compressed.

Another concern with the PS model and all-reduce is that they are designed to work on compute clusters. Hence, data must be moved from external sources to the cluster where the model is trained. In many cases, such centralization of data may be impossible due to privacy concerns, and legal restrictions, such as the European GDPR [23].

As a solution to these privacy concerns, federated learning (FL) was proposed. In federated learning, the gradients are calculated where the data is stored. For example, Google uses FL to train their Gboard keyboard's suggested queries [44]. As a result, information about the context of the query does never leave the user's smartphone. Instead, the model is trained locally with the user's data, and subsequently, the resulting gradients or model parameters are sent to a PS. However, in addition to the already mentioned drawbacks of the PS architecture FL comes with additional weaknesses. Firstly, gradients can be used to infer information about the training data [22]. Hence, the users must trust the central entity not to infer information from their gradients. Secondly, the training relies on a central entity, which means if it fails, then the other participants cannot make further training progress.

DL addresses these issues by removing the need for a central parameter server. For example, in algorithms such as decentralized parallel stochastic gradient descent decentralized parallel stochastic gradient descent (D-PSGD) [39], this is achieved by nodes sending their parameters to neighbors after one or more steps of local SGD. Each node then averages all the received models to get an improved model used in the subsequent SGD steps. Hence, the bandwidth bottleneck at the parameter server is removed, and models can continue to train even if one or more nodes fail.

One drawback of DL is that each node must now communicate more than in FL. It must send its parameters to all neighbors and receive from all neighbors. For large models with millions of parameters, this becomes quickly unfeasible. Hence, methods for reducing the amount of data to share are needed. Several methods, such as random subsampling [27] have been proposed. Some methods, such as CHOCO [33], go even further by proposing that each node saves a cached version of every neighbor's model. This cached version is then updated by compressing the changes since the last exchange. One proposed technique to compress these changes is sending the TopK changed parameters.

This thesis investigates the impact of several compression schemes on model convergence and proposes a combination of compression methods as a new communication efficient DL algorithm called Jwins++. It is a continuation of the Jwins [53] algorithm into which some work of this thesis was integrated. Jwins++ consists of four parts. The first is an importance score used to decide which parameters should be shared in every round. This score considers model changes due to local SGD steps and model averaging. The parameters with the TopK highest score are shared in each round, and subsequently, their scores are reset. The second method is the transformation of the model into the wavelet domain. Hence, the aforementioned score is tracked in the frequency domain of the model. The wavelet transformation affords two benefits. Firstly, it produces a high and low-frequency representation of the model, and secondly, it provides an approximation of the vector at different subsampling levels. The third method is the use of lossy zfp [41] compression on the TopK selected parameters in the wavelet frequency domain. The fourth method is a per-parameter averaging scheme, for which our experimental results show faster convergence and higher accuracies at low partial model sharing rates. Our experiments show that combining these four methods yields a compression ratio of up to 30x with little impact on the final accuracy.

The structure of this thesis is as follows. Section 2 provides background information on the core topics of this thesis, like machine learning, decentralized learning, and wavelet transformations. Section 3 gives an overview of the DecentralizePy [16] framework, which has been in development at SaCS for the past year. Many contributions have been made to the framework throughout this thesis. A complete list of these contributions is available at the end of the section. Section 4 presents the methodology used to run the experiments. Section 5 introduces the core parts of Jwins++ algorithm and evaluates them and the entire algorithm in several experiments. In Section 6, alternative communication methods are evaluated to test if random walks or dynamic graphs can reduce the amount of data that needs to be shared. A conclusion of the thesis is presented in the final Section 7.

2 Background

This chapter describes the core concepts needed to understand decentralized machine learning and our proposed algorithms. It starts with section 2.1 that gives a short primer on machine learning. The second section 2.2 focuses on distributed machine learning and related concepts such as federated learning and gradient compression. The third section 2.3 gives an introduction to decentralized machine learning, the topic around which this thesis evolves. Lastly, section 2.4 gives a primer on the discrete wavelet transformation.

2.1 Machine Learning Primer

Machine learning aims to find the global minimum of a d-dimensional function f over a dataset D. More formally we aim to optimize the following function:

$$\min_{w \in \mathbb{R}^d} F(w) := \frac{1}{|D|} \sum_{d \in D} f(w; d) \tag{1}$$

In the case of deep learning, the function f constitutes a neural network, i.e., a machine learning model consisting of a sequence of layers. Two elementary examples of such layers are convolutional layers and linear layers. These two layers build to the core of many neural networks, such as ImageNet [17] and LeNet [36]. A typical architecture consisting of these two models usually starts with one or several convolutional layers followed by linear layers.
A linear layer consists of a weight matrix $W \in \mathbb{R}^{m \times n}$, a bias $b \in \mathbb{R}^m$, and a non linearity $h : \mathbb{R}^m \to \mathbb{R}^m$. It takes an input $x \in \mathbb{R}^n$ and produces an output $y \in \mathbb{R}^m$. A mathematical description of a linear layer is given in equation 2.

$$y = \textit{linear(x)} := h(Wx + b) \tag{2}$$

A popular choice for the non-linearity is the rectified linear unit (relu) [47] function:

$$\textit{relu(z)} = \begin{cases} z & z \geq 0 \\ 0 & z \leq 0 \end{cases} \tag{3}$$

In the case of multi-layer neural networks, the d-dimensional input w of equation 1 represents all the parameters from all the layers. These parameters are also called the weights of the neural network. It can be thought of as a flattened and concatenated version of the parameters. For example, in the case of a linear layer from equation 2 the weights vector w would consist of the flattened weight matrix W with size $m \cdot n$ concatenated with the bias b to form a single vector of size $m \cdot n + m$.

2.1.1 Stochastic Gradient Descent

Finding a global minimum for the equation 1 is difficult, as there does not exist a method that guarantees to find the optimal weight w for all functions f. Nonetheless, neural networks are successfully optimized with stochastic gradient descent (SGD) [50, 6], which in general only converges to a local minimum of the function f.
SGD is an iterative algorithm that starts with a randomly initialized weight vector $w^{(0)}$ and proceeds to update the weight over T iterations with the following formula:

$$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla f(w^{(t)}; \xi^{(t)}) \tag{4}$$

The equation describes how at time step t, the gradient of f, multiplied by the learning rate η, is subtracted from the current solution candidate $w^{(t)}$. The gradient is evaluated for a sample

$\xi^{(t)}$ drawn uniformly at random from the dataset D. The result of equation 4 is a new candidate solution $w^{(t+1)}$.

Stochastic gradient has two significant drawbacks. Firstly, since the gradient calculation is based on a single sample, the resulting gradient has a high variance. Secondly, it is hard to parallelize SGD over multiple machines. Therefore, mini-batch SGD [5] is preferred over SGD in most machine learning tasks. It differs from SGD as the gradient is evaluated for a batch b of samples. These batches are created by splitting the dataset into $M = \lfloor \frac{|D|}{b} \rfloor$ mini-batches of size b. The remaining samples are either dropped, or they are used in a final smaller mini-batch. For a mini-batch $B_i = \{d_{i_0}, d_{i_1}, ..., d_{i_b}\}$ with $i \in [M]$, the Equation 4 can be expressed with mini-batches as follows:

$$w^{(t+1)} = w^{(t)} - \eta \cdot \frac{1}{|B_i|} \sum_{j=0}^{B_i} \nabla f(w^{(t)}; d_{i_j}) \tag{5}$$

The only difference to Equation 4 is that the subtrahend now consists of an average of gradients corresponding to the samples d_{i_j} in the minibatch B_i.

Mini-batch SGD is usually run for several passes over the dataset, called training epochs. For each of these epochs, a new set of mini-batches is created. The complete mini-batch SGD algorithm is summarized in Algorithm 1.

Algorithm 1 Mini-batch Stochastic Gradient Descent

```
Require: The number of epochs $E$, the batchsize $b$, the learning rate $\eta$, and the function $f$ with
    parameters $w$.
1: procedure MiniBatchSGD($w$, $B_j$)
2:     return $w - \eta \cdot \frac{1}{|B_j|} \sum_{k=0}^{|B_j|} \nabla f(w; B_j[k])$
3:
4: for $e \leftarrow 0$ to $E$ do
5:     $B \leftarrow$ new set of $M$ mini-batches of size $b$          ▷ Leftover samples are dropped
6:     for $j \leftarrow 0$ to $\lfloor \frac{|D|}{b} \rfloor$ do
7:         $w^{(e,j+1)} \leftarrow$ MiniBatchSGD($w^{(e,j)}, B[j]$)
8:     $w^{(e+1,0)} \leftarrow w^{(e,j+1)}$
```

Mini-batch SGD lends itself well to parallelization. The algorithm can be run across n machines, each calculating the gradient for an n-th of the mini-batch size b. These n parts can then be combined on a central node (Parameter Server)[15], which then updates the weight and distributes it to all machines.

2.2 Distributed Machine Learning

The algorithms presented in the previous section were designed to run on a single machine. However, current state-of-the-art neural networks have several billion parameters and are trained on billions of samples [9], which makes it unfeasible to train them on a single machine. For example, even training a moderately sized model such as ResNet-50 [26] on the ImageNet [17] dataset takes several days on a single GPU machine. Therefore, algorithms are needed that enable training on a cluster of machines. Distributed machine learning is the subdiscipline of machine learning that concerns itself with developing such algorithms.

Some assumptions made in section 2.1 do no longer hold for distributed systems. In particular, the dataset D can no longer be assumed to be available to all machines in its entirety. It is usually distributed among all the machines. Thus, for n machines, each machine $i \in [n]$ will have its own dataset D_i. If possible, the data is usually independent and identically distributed (IID).

A IID data split can be achieved by uniformly at random (without replacement) sampling the original dataset D. If this is not possible, then we speak of non independent and identically distributed (non-IID) data. A data distribution is, for example, non-IID if each machine only stores data corresponding to one or a few of the possible labels. However, it may also be non-iid if the data generated by the same source, e.g., a user's comments on an online messaging board, is only stored on one of the machines. Another assumption that no longer hold in distributed machine learning is that there is a single weight w. Each node $i \in [n]$ may have its own model w_i that differs arbitrarily from any other node's weight.

2.2.1 Parameter Server Architectures

A widely used concept in distributed machine learning is the parameter server architecture [38, 56, 15]. It consists of two distinct components: the parameter server and worker nodes. The job of the parameter server is to update the global parameters with the gradients that it receives from the worker nodes. Meanwhile, the worker nodes wait for the averaging on the PS to complete and subsequently download the most up-to-date model. The new model is then used to compute the gradient for a batch of data from their local dataset. Finally, they send the gradient to the PS and wait again for the new model weights. A detailed description of this training loop for both the parameter server and the worker nodes is given in Algorithms 2 and 3 respectively. Henceforth, this paradigm for training a neural network will be reference as **Centralized Stochastic Gradient Descent (C-SGD)**.

Algorithm 2 C-SGD: Parameter Server

```
Require: The number of global iterations $S$, connections to $n$ worker nodes, randomly initialized
    weight vector $w = w_0$ of to the machine learning model $f$.
1: for $s \leftarrow 0$ to $S$ do
2:     wait for all worker nodes to pull the current model $w$        ▷ synchronizes all nodes
3:     declare $G[n]$
4:     for $i \leftarrow 0$ to $n$ do
5:         $G[i] \leftarrow$ receive gradient $g_i$ from node $i$        ▷ can be asynchronous
6:     $w \leftarrow (w - \frac{1}{n}\sum_{k=0}^{n-1} G[k])$
```

Algorithm 3 C-SGD: Worker Node i

```
Require: The number of global iterations $S$, the learning rate $\eta$ and a connection to the pa-
    rameter server PS.
1: procedure GetGradientSGD($w_i$)
2:     $\xi_i \leftarrow$ random sample drawn from $D_i$
3:     return $\eta \cdot \nabla f(w_i; \xi_i)$
4:
5: for $s \leftarrow 0$ to $S$ do
6:     $w_i \leftarrow$ download the current global weight from the PS
7:     $g_i \leftarrow$ GetGradientSGD($w_i$)
8:     send $g_i$ to the PS
```

In the algorithm 3, a gradient corresponding to a single sample is produced, i.e., a single step of SGD is performed. The gradients are combined on the parameters server, meaning the effective update resembles the mini-batch update of algorithm 1 with batch size n. An alternative implementation of algorithm 3 could use a local mini-batch of size b instead of a random sample. In this case, the global batch size would be $n \cdot b$.

There are many more variations on C-SGD. For example, Downpour SGD[15] is an asynchronous implementation of the parameter server model. Meaning that the worker nodes are not synchronized after they computed their gradients. Instead, they run independently, and the parameter server updates the global model as soon as it gets a new gradient. Downpour SGD also allows nodes to do local SGD steps [43]. For local SGD steps, the worker node does not send its gradient to the PS. Instead, it updates its model w_i locally and accrues the gradients. At the end of local training, the worker node sends the accrued gradient to the PS and overwrites their model w_i with the global model w. These local SGD steps help to reduce the number of communication rounds, which reduces the bandwidth requirement for the parameter server.

2.2.2 Federated learning

Federated learning was introduced by Brendan McMahan et al. at Google [45]. It was conceived to train models without moving sensitive user data from smartphones to server clusters by training the model where the data is.

Two algorithms were proposed FedSGD and FedAvg. These algorithms work in conjunction with a parameter server, which selects a subset of nodes to participate in the training process in every round. These nodes receive the newest model and then start the training process. In the case of FedSGD, the node will calculate a gradient over its entire local training set and send it back to the parameter server, where the gradients are averaged and applied to the current model. For FedAvg, the node performs several local SGD steps and several passes over its local data before it sends back the model to the PS, where it gets averaged with all the other participants' models.

Federated learning still relies on a centralized entity to run the parameter server, which is a problem as it may use information gained from gradients to infer information about the nodes' data [22]. There exist methods to mitigate these privacy concerns. For example, distributed selective SGD [54] uses gradient sparsification to send the gradient's most changed parameters (or a subsampled set of gradient values larger than a threshold) and combines it with differential privacy.

2.2.3 Gradient Compression

The parameter server architecture introduced in Section 2.2.2 has the problem that every node must both upload their gradients and download the new model. These two activities induce a heavy bandwidth load at the parameter server. Therefore, much research focused on reducing the amount of data that the PS sends or receives. Gradient compression focuses on the latter part. It is applied on the worker nodes to the gradients and aims to reduce their size significantly. Another advantage of decreasing the number of shared values is reducing the communication latency. For FL, these savings are even more potent as mobile phones are most likely not in networks with symmetrical bandwidth. So, reducing the upload size, i.e., the gradient, is essential.

There are two widely used techniques to reduce the size of the gradients. The first one is gradient sparsification. For example, random subsampling [34] can be used to reduce the number of values exchanged. It uses a random mask to select the parameters. Since the random mask can be recreated with only a seed, there is no need to share additional metadata like the indices of the shared parameters. Another technique for sparsifying the gradient is to send either only the gradients with absolute values larger than a threshold [57, 54] or by sending the TopK highest values [19, 40, 54].

The second technique is gradient quantization. Gradients can, for example, be represented as 16-bit [24], or even 8-bit [18] floating point numbers. Even a 1-bit representation is possible [51]. For such high compression ratios, error compensation [51] is applied, i.e., the difference between the actual gradient and the quantized gradient is accumulated. This accumulated difference is then added to the gradient before the quantization in all subsequent rounds. It ensures that all the gradient information will be transmitted eventually. Accumulation can also be combined with gradient sparsification [40].

Another low-bit quantization scheme is QSGD [1]. It uses the Elias [20] integer compression algorithm to compress the quantized values further by compressing them directly and by compressing the distance between non-zero values, i.e., values with the quantized value of zero are skipped.

2.3 Decentralized Machine Learning

Decentralized machine learning differs from centralized machine learning in two significant ways. Firstly, DL is designed to train models where the data D is stored. Therefore, data does not need to be moved to a server cluster, which is especially useful for sensitive personalized data as it alleviates privacy concerns about sharing the data with third parties. Secondly, in DL there is no need for a centralized coordinator like the parameter server in C-SGD. Instead, the nodes communicate directly and work to optimize the machine learning model jointly.

There are several proposed approaches for decentralized learning, such as algorithms based on distributed alternating direction method of multipliers (ADMM) [28] and algorithms based on distributed consensus averaging[?]. This thesis will only consider algorithms belonging to the latter category. The principal algorithm of that category is called D-PSGD [39]. A pseudocode implementation of it is given in Algorithm 4. It is essentially a combination of two algorithms. The first algorithm is used to achieve a global average consensus over the values held by each node [64]. A more detailed explanation of global average consensus is given in Section 2.3.1. The second algorithm is a simple step of SGD using a sample of the node's data. However, it also works mini-batch SGD.

Algorithm 4 Decentralized Parallel Stochastic Gradient Descent

```
Require: The algorithm is run on all $N$ in parallel. On every node $i \in [N]$ initialize $x_{0,i} = x_0$.
    The weight matrix $W$, local dataset $D_i$, the learning rate $\eta$, the number of communication
    steps $T$.
 1: procedure DistributedAverageConsensus($x_i$)
 2:     send $x_i$ to all neighbors
 3:     receive $x_j$ from all neighbors for $j \neq i$        ▷ Synchronizes the node with its neighbors
 4:     return $\sum_{j=1}^{N} W_{ij} x_j$
 5: procedure LocalSGD($x_i$)
 6:     $\xi_i \leftarrow$ random sample drawn from $D_i$
 7:     return $x_i - \eta \cdot \nabla f(x_i; \xi_i)$
 8:
 9: for $t \leftarrow 0$ to $T$ do
10:     $x_i^{t+\frac{1}{2}} \leftarrow$ LocalSGD($x_{t,i}$)
11:     $x_i^{t+1} \leftarrow$ DistributedAverageConsensus($x_i^{t+\frac{1}{2}}$)
```

2.3.1 Distributed Consensus-Averaging

Distributed consensus averaging [63, 64] is an algorithm to calculate a global mean value in a communication topology given by graph $G = (V, E)$ with $N = |V|$. Each node $V_i \in V$ with $i \in [N]$ starts with an initial value x_i^0 and iteratively updates its with the values it receives from its neighbours $M_i = \{k : (k, i) \in E\}$. The iterative update rule is:

$$x_i^{t+1} = \sum_{j \in M_i \cup \{i\}} W_{ij} \cdot x_i^t \tag{6}$$

The matrix $W \in \mathbb{R}^{N \times N}$ is called the weight matrix. Together with the vector of all node values $x^t \in \mathbb{R}^N$ such that $(x^t)_i = x_i^t$, Equation 6 can be reformulated as a matrix-vector multiplication:

$$x^{t+1} = Wx^t \tag{7}$$

In this form the global average $\bar{x}$ is given by:

$$\bar{x} = (\frac{1}{N}\mathbf{1}^T\mathbf{1})x^0 \tag{8}$$

In other words global average consensus converges in one step on a fully connected graph. For other graphs, Xiao et al. [63] showed that the iterative update from Equation 6 converges toward the global average $\bar{x}$ if and only if these three conditons hold:

$$\mathbf{1}^T W = \mathbf{1}^T \tag{9}$$

$$W\mathbf{1} = \mathbf{1} \tag{10}$$

$$\|W - \frac{1}{N}\mathbf{1}^T\mathbf{1}\| < 1 \tag{11}$$

Equations 9 and 10 together state that both the rows and columns of the matrix W sum up to 1, i.e., the matrix is doubly stochastic. In Equation 11 the matrix norm $\|\cdot\|$ is the spectral norm, i.e., the square root of the highest absolute singular value. These conditions guarantee that the largest eigenvalue of W is one and all others lie between -1 and 1 [63].

If these conditions hold then the following inequality holds:

$$\|x^{t+1} - \bar{x}\|_2 < \|W - \frac{1}{N}\mathbf{1}^T\mathbf{1}\|\|x^t - \bar{x}\|_2 \tag{12}$$

If the matrix W is symmetric then Equation 13 can be expressed as [33, 63]:

$$\|x^{t+1} - \bar{x}\|_2 < (1 - \rho(W))\|x^t - \bar{x}\|_2 \tag{13}$$

with ρ denoting the eigengap, i.e., the difference between the largest and second largest eigenvalue of W.

One weight matrix fulfilling all these conditions is the metro-hastings weight matrix [64]

$$(W)_{ij} = \begin{cases} \frac{1}{\max(d_i,d_j)+1} & (i,j) \in E \land j \neq i \\ 1 - \sum_{linM_i} \frac{1}{\max(d_i,d_l)+1} & j = i \\ 0 & else \end{cases} \tag{14}$$

with d_i and d_j denoting the degree of nodes V_i and V_j.

2.3.2 Model compression in Decentralized learning

The D-PSGD algorithm has been extended to work with compressed communication by different algorithms. Difference compression D-PSGD (DCD-PSGD) [58] is one such algorithm. It makes use of local replicas, which are copies of a node's model that are stored on its neighbors. They are used during the consensus averaging step instead of received models in D-PSGD. They are updated with the compressed model change due to local training, i.e., in Algorithm 4 the difference $x_{t+\frac{1}{2},i} - x_{t,i}$. DCD-PSGD works well for unbiased compression like random subsampling. The replicas are kept in sync because the nodes reset their own model weights to the value of the replica at the end of the round. CHOCO [33] is an algorithm similar to DCD-PSGD, but it works with biased compression techniques. Its replicas are no longer exact copies but approximations. At the end of a training round, each node compresses the difference between its current model and its replicate and sends this compressed difference to its neighbors. Additionally, CHOCO uses a modified version of consensus averaging with its own learning rate that needs to be tuned. Error-Compensated Sparsification D-PSGD (ECSD-SGD) [60] is another algorithm that is based on DCD-PSGD. It accumulates the compression error that is discarded in DCD-PSGD and adds it in subsequent updates to the gradient for the local SGD step. It uses the same modified version of consensus averaging as CHOCO and also works for biased compression techniques. As all these methods compress model differences, they can use the gradient compression methods introduced in section 2.2.3.

Other approaches for model compression that do not need replicas have been used too. For example, random subsampling and model partitioning has been used by Hegedűs et al. [27] in their decentralized learning framework. The model is split into parts for partitioning, and when the partial model is shared, a part is randomly selected. On the other hand, random subsampling chooses arbitrary parameters from the entire model.

2.4 Wavelet

The discrete wavelet transform (DWT) [10] is a frequently used preprocessing step applied to digital signals before they are compressed. It has two characteristics that make it popular as a preprocessing step. Firstly, it decomposes the signal into a low and high-frequency representation. Secondly, it can decompose a signal into a collection of representations at multiple resolutions. For example, the JPEG2000 standard [55] exploits both properties in its image compression pipeline. Since the DWT is a preprocessing step, the actual compression happens by compressing the outputs of the transformation. In JPEG200, quantization is used as the compression scheme. The discrete wavelet form represents a signal as a combination of a scaling function ϕ and its corresponding wavelet function ψ:

$$f(t) = \sum_{k=-\infty}^{\infty} c_{j_0}(k)\phi_{j_0k}(t) + \sum_{j=j_0}^{\infty}\sum_{k=-\infty}^{\infty} d_j(k)\psi_{j_0k}(t) \tag{15}$$

With $\phi_{jk} = 2^{j/2}\phi(2^j t - k)$ being the scaling function scaled by j and translated by k. The definition for the scaled and translated wavelet function ψ_{j_0k} is analogous. Equation 15 shows how the signal is decomposed into two parts. The leftmost sum represents a low-resolution representation of the signal f with the scale function scaled by j_0. The two sums on the right-hand side represent the second part. They describe the missing high-frequency information using wavelet functions at all scales $j > j_0$. The wavelets at a low scale capture more frequency information about the signal, while wavelets at a high scale capture more spacial resolution. Therefore, the DWT gives a good spatial and frequency resolution. Hence, the DWT is better suited for signals whose frequencies change over time than the discrete fourier transformation, which gives gives only a frequency decomposition of the signal. The scale coefficients $c_{j_0}(k)$ and wavelet coefficients can be calculated by the inner product of the scale function respectively the wavelet function with the signal f.

$$\begin{aligned} c_{j_0}(k) &= \int f(t)\phi_{j_0k}(t)dt \\ d_j(k) &= \int f(t)\psi_{jk}(t)dt \end{aligned} \tag{16}$$

The Equation 15 seems impractical with its use of infinity sums. However, both the wavelet and scale functions are only non-zero for a short interval in practice. For example, the Sym2 wavelet is only defined in the interval $[0, 3]$. Hence, if the signal is finite, then k is also finite. Hence, two of the three infinite sums are already eliminated, which leaves the infinite sum for the scaling factor j. This sum can be made finite by assuming that the sampled signal f(t) already corresponds to the wavelet coefficients at a high scaling factor. In other words, instead of starting with the low-frequency representation of the signal (first sum of Equation 15), an efficient DWT algorithm starts with a representation of the signal with a wavelet function at a sufficiently high scaling factor j_{max} to correctly represent the signal. It then works itself downwards towards a low-frequency representation at scale j_0. The outlined approach is what a filter bank-based [59, 11] implementation of the DWT does.

$$\begin{aligned} \psi(t) &= \sum_n h_1(n)\sqrt{2}\phi 2t - n \\ \phi(t) &= \sum_n h(n)\sqrt{2}\phi 2t - n \end{aligned} \tag{17}$$

Therefore convolving the original signal with filters h_1 and h will represent the signal in the space of the wavelet and scaling functions at half the frequency. Subsequently, both representations are subsampled by a factor of two. Subsampling is possible because, according to the Nyquist-Shannon sampling theorem [52], only half the samples are needed to represent a signal of half the frequency. The results are an approximation vector a, produced using the low-pass filter h, and a detail vector d, produced using the high-pass filter $h1$. The filter bank algorithm can repeat these steps on the approximation vector a. The result then is a multi-level decomposition consisting of several detail vectors d_i for $i \in [0, max\_level]$ and a single approximation vector a. The entire filterbank algorithm is represented in Algorithm 5. It shows an additional step at line 3, the signal is padded to enable the inverse DWT, which takes a list of detail vectors and the approximation and converts it into a signal. Both the DWT and the inverse DWT have a complexity of $O(N)$.

Algorithm 5 Descrete Wavelet Transformation

```
Require: Highpass fiter $hi$; Lowpass filter $lo$; level
 1: $levels \leftarrow []$ $a \leftarrow x_i$
 2: for $l \leftarrow 0$ to $level$ do
 3:     $e_i \leftarrow reverse(a[1 : hi.length]) + +a + +reverse(a[-hi.length + 1 :])$
 4:     $a \leftarrow \mathbf{0}$
 5:     $d \leftarrow \mathbf{0}$
 6:     $c \leftarrow 0$
 7:     for $i \leftarrow hi.length + 1; i < e_i.length; i \leftarrow i + 1$ do
 8:         for $j \leftarrow 0; 0 < hi.length; j \leftarrow j + 1$ do
 9:             $a[c] \leftarrow a[c] + e_i[i - j] \cdot lo[j]$
10:             $d[c] \leftarrow d[c] + e_i[i - j] \cdot hi[j]$
11:         $c \leftarrow c + 1$
12:     $levels.append(d)$
13:     if $l == level - 1$ then $levels.append(a)$
14: return $cat(levels)$
```

The discrete wavelet form was considered as possible method for compressing machine learning models [13, 31], but these projects needed good estimations for a compression factor, which wavelet based compression does not provide. However, related transformations like the fast fourier transform has been used in machine learning as a prepossessing step to gradient compression with TopK selection [61].

3 DecentralizePy

The DecentralizePy [16] framework for decentralized machine learning has been in development at EPFL's Scalable Computing Systems Laboratory (SaCS) since November 2021. It was principally developed by Rishi Sharma with contributions from other members of SaCS and project students. Section 3.1 gives an overview of the framework. A more in-depth look at datasets and models supported by the framework is presented in the subsequent Section 3.3. Finally, this section concludes with a rundown of the contributions that were part of this Master's Thesis.

3.1 Framework Overview

DecentralizePy is a Python 3 framework that aims to simulate large-scale decentralized learning tasks on CPU clusters. It is run on all or a subset of machines in the cluster and simulates additional virtual nodes on every participating machine. Therefore, it is capable of running hundreds of nodes across several machines.

Figure 1 shows the folder structure of DecentralizePy alongside some important modules. The project is split into an evaluation sections *eval* and the implementation *src*. The evaluation part contains all configurations necessary to run the experiments of this thesis, while the implementation part contains the actual source code. Both these parts are explained in more detail in the following sections.

3.2 Evaluation Files

The evaluation section contains all the configuration files necessary to run DecentralizePy on a server cluster or a single machine. In order to make every machine in the cluster aware of which machines participate, a network configuration file named *ip_ addr_ xMachine.json* must list the IP addresses of all the participating machines. Several of these files are included that correspond to different combinations of machines on our server cluster. The x in the file name is used to indicate the number of machines.
The folder *step_ configs* contains an exhaustive list of INI configuration files. These files enable a framework client to specify exactly how the models are trained. Among other options, they offer the possibility to choose a communication protocol, the optimizer for training the network, and the model sharing strategy.

The evaluation section also includes the main file to start the framework. It is called *testing.py* and must be executed on every machine. Several input arguments need to be provided to it. One of them is an aforementioned INI configuration file. The other arguments are orthogonal to options specified in the INI file and are usually options that a client may want to change in every run, like the logging directory or the number of training iterations. A complete list of all its arguments is given in the *src/decentralizepy/utils.py* file.

For all experiments, we used bash run files contained in the directory to start the execution of DecentralizePy. These bash files run *testing.py* with the correct arguments for the specified experiments. There are two types of them. The grid-search run files with the patterns *run_ grid_ {dataset}.sh* and the actual run files for the experiments with the patterns *run_ xtimes_ {dataset}_ {experiment}.sh.* Both of them take the same four inputs: A absolute path to a home directory, the absolute path of the python *bin* bin folder, the path to the logs folder relative to the path of the first argument, and the name of the network configuration file. The network configuration file must be copied to a folder named *configs/* inside the specified home directory. For our experiments, this home directory is an NFS folder. Hence, all the outputs are already centralized and available for post-processing. Each machine writes its logs into a folder tagged with its machine id, which is determined by the position of the machine's IP address in the network configuration file.

```
decentralizepy
├── eval
│   ├── step_configs
│   ├── ip_addr_xMachines.json
│   ├── run_xtimes.sh
│   ├── testing.py
│   ├── plot.py
│   └── ...
└── src
    └── decentralizepy
        ├── communication
        │   ├── Communication.py
        │   └── ...
        ├── compression
        │   ├── Compression.py
        │   └── ...
        ├── datasets
        │   ├── Dataset.py
        │   └── ...
        ├── graphs
        │   ├── Graph.py
        │   └── ...
        ├── mappings
        │   ├── Mapping.py
        │   └── ...
        ├── models
        │   ├── Model.py
        │   └── ...
        ├── node
        │   ├── Node.py
        │   └── ...
        ├── sharing
        │   ├── Sharing.py
        │   └── ...
        ├── training
        │   ├── Training.py
        │   └── ...
        ├── utils.py
        └── train_test_evaluation.py
```

Figure 1: Top level overview of the DecentralizePy project

The script file must be adjusted manually for other details such as the number of machines, the number of nodes per machine, or the used INI files.

Grid-search Scripts. The grid search scripts contains lists of values for three hyperparameters that were tuned for all our models:

- **Learning rates**: A list of learning rates **lr** for the SGD optimizer.
- **Batchsizes b**: A list of mini-batch sizes **b**. The mini-batch size determines the number of samples used in a single mini-batch SGD step. It also determines how many iterations there are in a pass over the local dataset.
- **Communication rounds per global epoch r**: A list of values specifying the number of communication rounds **r** in a single pass over the global dataset. For a dataset of size $|D|$ and N nodes, every node will do $\frac{|D|}{r \cdot b \cdot N}$ local SGD steps before sending its model parameters to the neighbors.

The script will evaluate all combinations of the provided values. For example, for three learning rates, four batch sizes, and two different r-values, the script will test all $3 \cdot 4 \cdot 2 = 24$ combinations. For each of these combinations, a new log folder is created containing the experiment's name derived from the INI configuration file name, the learning rate, batchsize, r-value, and the current time.
Furthermore, the INI files need to be modified for every combination of these values as the learning rates, batch sizes, and the number of iterations per communication round are defined in them. Hence, a copy of the INI file is created and modified using crudini [8], a python library for modifying INI files. One last thing that the script handles is setting the test and trainset evaluation interval to the number of iterations in a global epoch.

Experiment Scripts. The experiment bash scripts are very similar to the grid-search scripts. However, instead of evaluating every possible combination of hyperparameters, they evaluate a list of INI files for a list of provided seeds for a pseudorandom number generator. The seeds impact the model evaluation, the shuffling of the training batches, and the initial data distribution.

Plotting Scripts. The main plotting script is called *plot.py.* It will read the result files stored in the log directories and crate plots for train and test loss, test accuracy, and other metrics such as the total bytes shared. The script reads the results for every node and calculates the average value and standard deviation. These results are stored in separate CSV files that are used by other plotting scripts, like *plotting_std.py*, which creates the plots used in this thesis. The script also allows for replotting with either the global epochs or iterations as the x-axis. Lastly, *plot_percentile.py* plots how many times every percentile of parameters has been shared for experiments that use partial model sharing.

3.3 Implementation

The *src* section of the DecentralizePy framework holds the core implementation. The framework is designed to be highly customizable. It consists of different parts that are introduced in the following sections. Each part may have several implementations that can be selected and combined in the INI configuration files, except the *node.py* module, which is the framework's core.

3.3.1 Communication

The abstract *Communication* class exposes communication functions to other parts of the framework. The interface consists of six functions: two functions for establishing and closing communication links with all neighboring nodes, another two to send and receive messages, and lastly, two functions for encrypting and decrypting data. There are three implementations of the *Communication* interface currently.

TCP. The file *TCP.py* contains an implementation of the *Communication* interface based on the ZeroMQ messaging protocol [4], which is in turn based on the TCP protocol. The ZeroMQ library is designed to offer a uniform interface for both intra and inter-machine communication. Therefore, it is very well suited to handle both the communications between simulated nodes on the same machine and nodes on different machines.
Since each machine runs many virtual nodes, a TCP port is needed for every node. The ports are calculated based on a unique rank that each node is assigned. The actual communicaton links between the nodes are implemented with a ZMQ router for receiving messages and a ZMQ dealer per neighbor to send messages. The latter is created when the function *connect_ neighbors* is called. The opening of a connection is confirmed by sending a hello message. A node waits for hello messages from all its neighbors before it finishes the connecting phase. It also waits for a bye message from all neighbors when *disconnect_ neighbors* is called.
Before messages are sent, they are encoded with Python's pickle function. If a compression module is specified in the INI file, then the *encrypt* function will compress both the list of indices and the parameters with their respective compression functions. For this to work, the message must be a dictionary that contains the keys "indices" and "params" for the indices and parameters, respectively. The decryption works analogously.

TCPRandomWalk. The *TCPRandomWalk.py* module implements an asynchronous version of the *Communication* interface that also handles random walk messages. It consists of two classes: *TCPRandomWalk* and *TCPRandomWalkInternal.* Both of them implement the *Communication* interface. However, only the former is exposed to the other parts of the framework, while the latter is initiated in its own process once the *connect_ neighbors* function of *TCPRandomWalk* is called. Since it runs in its own process, it can send and receive messages even while the node is training the model. The two classes communicate via two shared multiprocessing queues to store the received messages and the messages scheduled to be sent. A shared memory flag is also used to notify the internal version when *disconnect_ neighbors* has been called. If the flag is set to zero, then the processing loop of the internal version stops, and the neighbors are disconnected.The internal class operates similarly to the *TCP* implementation. It differs by using a processing loop that checks every iteration if the router has received a message and if there are messages to send in the send queue.

In addition to normal messages, *TCPRandomWalk* also handles random walk message. These messages are immediately forwarded to another neighbor in addition to being delivered to the node. However, they are only forwarded to a randomly selected neighbor if they still have fuel left, which is tracked by an additional entry in the message's dictionary. The initial amount of fuel corresponds to the length of the random walk.

In order to limit the performance overhead of the asynchronous communication, the processing loop uses an exponential back-off scheme. Hence, if it did not process any messages in a while, it will preserve resources by sleeping for longer intervals.

TCPRandomWalkRouting. The *TCPRandomWalkRouting* supports asynchronous communications as well as connecting and disconnecting neighbors. The connection process works by reappropriating the random walk messages. These messages no longer contain the parameters. Instead, they contain the sender's id and the communication round during which they were sent. The sending of these random walk messages is left to the client of the interface, i.e., an implementation of the *Sharing* interface (See Section 3.3.8). A node establishes a connection to the originator of a random walk message if its fuel count has reached zero. Therefore, only one new connection is established for every random walk message. The establishment of connections is complicated. The final recipient of the random walk message sends a hello message together with the current iteration number to the source of the random walk. It is then blocked from proceeding to the next round until a response is received. If the response is tagged with the same communication round, then the neighbor is added, and the model parameters are exchanged. If the response is tagged with a future round, then the node can proceed to the next round, and the connection is established once that round is reached. On the other side, if the source of the random walk receives a hello tagged with a higher round, then it will wait with its response until it reaches that round. If it is in the same round or ahead, it will send a response immediately and has to wait for the other node to catch up.

The number of neighbors a node is permitted to have is bounded. Once the upper bound is reached, a random neighbor is selected, and a bye message is sent. If this neighbor is at the lower bound, it responds with a nobye message. If this is the case, a new disconnect attempt is started with a new neighbor in the next round. Otherwise, the neighbor responds with a bye message, and they remove each other as neighbors at the end of the round. If the neighbor is behind, it will wait with processing the bye until it catches up. The originator of the bye will have to wait until the neighbor is caught up. Finally, in case the neighbor is ahead, it sends back a bye with its current round, and the bye will be postponed to the end of that round. A particular case may arise when a node receives a bye for a future neighbor. In that case, the neighbor is put on both the future neighbors and bye list. Two additional exceptional cases arise when two nodes start a hello or bye with each other independently. These cases are automatically handled, as each node assumes the bye or hello was a response to the bye or hello initiated by them.

3.3.2 Compression

The *Compression* interface consists of two pairs of functions used by the communication interface's *decrypt* and *encrypt* functions. The first pair, *compress* and *decompress*, is used on the indices and the second pair, *compress_float* and *decompress_float*, is used to compress the floating point model parameters. There are several implementations of the Compression interface available. All but one use the Elias gamma [20] compression to condense the indices. The only one not using it uses lz4 [14]. For the floating point compression, there exist many implementations. For example, a version uses 16-bit floating point numbers instead of 32-bit floats. Another version uses the PyTorch [49] quantization to produce an 8-bit quantization. Finally, there are also several versions based on the fpzip [42] and zfp [41] floating point compression algorithms, which besides lossless compression, also allow for lossy compression. These compression techniques are introduced in more detail in Section 5.3 and evaluated in Section 5.5.7

Elias Gamma Compression. The Elias gamma [20] compression is a simple universal variable length encoding for binary numbers. It represents any binary number b by prepending it with $\lfloor log_2(b) \rfloor$ zero bits. For example, the Elias gamma representation of 13 consists of three prepended zeros followed by the number 13 expressed in binary, i.e., 0001101. Hence, the Elias gamma compression is very space efficient for small numbers. However, the index of a vector consists of large numbers. Therefore, the sorted index is turned into a difference vector. Since the difference vector is insufficient to reconstruct the indices vector, the first index plus one is prepended before applying Elias compression. Adding one is necessary as the Elias compression only works for integers larger than zero. For example, to compress the indices [1,3,4,7], it is transformed into the following difference vector [2,1,3], and the 1+1 is prepended to yield [2,2,1,3]. Finally, the Elias gamma compression is applied to all the numbers in the vector.

The decompression works in reverse. First, the reverse Elias compression is applied, then one is subtracted from the first index, and finally, the cumulative sum is calculated to yield the original indices vector.

This method of compressing the metadata is similar to the method outline in QSGD [1].

FastVarInt Package. The Elias gamma compression is calculated using the open-source FastVarInt python package [62] developed as part of this thesis. The package is based on the *scikit_build_example* provided by PyBind11 [30], a project that enables the creation of Python bindings for C++ code. We chose to implement the Elias compression in C++ as it requires bit manipulation, which is slow in Python. The package is available for download via the PyPI [30] package manager. However, it is currently only working on Linux systems.

3.3.3 Datasets

The abstract *Dataset* interface provides two functions, one to get the trainset and the other to get the testset. The interface is used to implement many datasets and models from the Leaf [12] Benchmark for decentralized settings. These datasets are split up into data generated by users. Thus, their data distribution is non-IID. However, their concept of a user is not the same for all datasets. For the Reddit dataset, the users are the actual commenters that wrote the posts, while for the Shakespeare dataset, the users are the characters of Shakespeare's plays. Therefore, there are stark differences in how non-IID these data sets are. Nonetheless, all implementations of the Leaf dataset use this user partition by default. Leaf provides a JSON file per user, which either contains the data directly, like for Femnist, Reddit, and Shakespeare or contains just a reference to the data, like for the Celeba dataset. These user files are equally distributed to each node by the *DataPartitioner*. In addition to the four Leaf datasets, DecentralizePy implements Movielens [25] and Cifar10 [35]. The former is not used in this thesis.

Femnist. The femnist dataset consists of 734 463 images of handwritten letters and numbers, i.e., there are 62 classes. The images were written by 3597 users and are partitioned according to these users. The testset consists of another 83 388 samples from these users that are not in the trainset. The images of the train and testset are black and white 28 by 28 images that are preprocessed and stored in the JSON files as a floating point array.

There are three models available for Femnist, a logistic model consisting of 48 608 parameters, a CNN model consisting of 1 690 046, and a downscaled version of Resnt18 that consists of 2 816 498 parameters.

The CNN model starts with a 2d convolutional layer using a 5x5 kernel, a padding of two, and 32 output layers. The relu non-linearity is used to transform the outputs, followed by a pooling layer with windows size two and stride two. The same setup is repeated with the CNN layer's output set to 64. The two final layers are two linear layers separated by a relu activation function. The output size of the hidden layer is 512 and the one of the final layer is 62.

The Resnet18 model is based on the torchvision implementaiton. Its input layer has been reduced to accept one layer instead of the intended three for color images. Additionally, the output size of all the convolutional layers has been halved.

Celeba. The Celeba trainset consists of 63 741 images of 2361 celebrities, with every celebrity at least having five images. These images are labeled based on whether they are smiling. A JSON file exists for each celebrity representing the users of this dataset. These files store image ids corresponding to the images stored in a separate folder. The testset consists of an additional 7097 of these celebrities' images that were not seen during the training. The images are stored as 84 by 84 RGB in PNG format.

There is one model available for Celeba. It is a CNN model that consists of four CNN layers. Each layer consists of a convolution with an output size of 32, a kernel of size three, and padding that keeps the output size the same as the input size. Each convolutional layer's output is transformed with the relu non-linearity and subsequently pooled by a max pooling layer with windows size two and stride two. The final layer is linear with an input size of 800 and two outputs. In total, there are 30 242 parameters.

Reddit. The Reddit trainset consists of 70 642 samples that correspond to comments of 816 Reddit users. The Reddit dataset is designed for next-word prediction. A sequence of 10 words is used to predict the next word. The words are transformed to a token id in a preprocessing step. In total, there are 9999 unique tokens. The testset consists of another 24 961 samples that are not included in the training set.

One model is available for the Reddit dataset. It is an RNN model with an embedding of size 200 for the tokens. It first maps the tokens into the learned embedding space. Tokens represented in this embedding space are inputs for a two-layer LSTM network. Each layer consists of an LSTM cell with a hidden state of size 256. The output of the second layer is used as the input for a linear layer, which feeds its relu transformed output of size 128 into a final linear layer with output size 9999. In total, the model consists of 4 317 895 parameters.

Shakespeares. The Shakespears dataset consists of the dialog of characters in Shakespeare's plays. We subsampled the dialog from 97 fictional characters that had between 800 and 1250 samples in the original dataset. For this dataset, the fictional characters are considered the users by which the data is split. The goal of the Shakespeare dataset is the next symbol prediction for an input of 80 symbols. There are 92 possible symbols. Lastly, the test set contains 356971 samples that have not been seen during training. From these samples, every fiftieth is selected. There is a single model for the Shakespeare dataset. It is an RNN similar to the one used for the Reddit dataset. It also uses an embedding, but it is of size eight and is used as input to a two-layer LSTM with a hidden layer size of 256. However, unlike the Reddit dataset, the hidden state corresponding to every entry of the input sequence is used for the subsequent linear layer. Hence, the linear layer's input is of size 256*80 = 20 480, and its output has a size of 92 (the vocabulary size). In total, the model consists of 2'683'708 parameters.

Cifar10. The Cifar10 [35] dataset has a trainset with 50 000 images. These are 32x32 color images, each belonging to one of ten classes. In order to get an non-IID data distribution, all images are sorted by their label. Subsequently, every node samples one or more random subsections from the data. We refer to these random sections as shards. Hence, not all data will be used, and some images may be used on more than one machine. The testset consists of another 10 000 images that do not appear in the trainset.
There are three models available. The first one is a custom CNN model, which is not used in this thesis. The second model is inspired by LeNet [36]. It consists of three CNN layers. Each layer consists of a convolution layer with kernel size five and padding that preserves the input size. The output sizes are 32, 32, and 64. The outputs are normalized using group normalization, and the relu non-linearity is subsequently applied. The final step of all layers is max-pooling with a window and stride of two. These convolutional layers are followed by a single linear layer with ten outputs, one for each label class. All these layers add up to 89 834 parameters. The third model is Resnet20 with 11 191 262 parameters.

3.3.4 Graphs

The *graphs* directory contains a set of modules for creating and manipulating graphs. The base class *Graph* contains functions for adding nodes and edges to a graph. It also enables a graph to be written to a file in either the adjacency or edges representation. Other functions provide helpful utility, such as making the graph connected or returning a list of neighbors for a specific node.
Other modules in the directory provide objects to create specific graphs, such as fully connected graphs, ring graphs, star graphs, regular graphs, and small world graphs. A star graph has a central node that is connected to all others. A regular graph has a fixed degree for all nodes with random edges connecting them. A small world graph is similar to a Erdős-Rényi random graph [21]. It has random edges and degrees.

3.3.5 Mappings

The *Mapping* interface provides utility functions for working with the two ids that every simulated node has. These two ids are the local rank, i.e., a unique number associated with every simulated node on a machine, and the global machine id. These ids start from 0 and increase by an interval of one, e.g., for six global machines, the maximum machine id would be 5. The most important function is *get_uid*. It returns a unique id for every node by combining the local rank and the machine id. Its only implementation is in the *Linear* class, and it assigns the uid with the following formula:

$$uid = mid \cdot n + localrank \tag{18}$$

with *mid* being the machine id and n for the number of processes per machine.

3.3.6 Models

The *Model* class implements a wrapper for the *torch.nn.Module* class. This wrapper is the superclass for all models listed in Section 3.3.3, implemented in the DecentralizePy framework. It contains additional properties that are needed by the framework. For example, the field *shared_parameters_counter* is used to count how many times a parameter has been shared, and the field accumulated_changes is used to track the accumulated model changes due to training and model averaging. Additionally, there are some useful utility functions like *get_weights* that returns a flat vector representation of the model's parameters.

3.3.7 Node

The *Node* class is the core of the DecentralizePy framework. It is initiated several times by the *eval/testing.py* stript to create virtual nodes. These virtual nodes are executed using the spawn function of the PyTorch [49] multiprocessing module. It allows the spawning of several processes that propagate errors and terminate automatically in case another process crashes. The node object needs several parameters for its initialization. The documentation in the *Node.py* module gives a good description of all these arguments. The most important argument is the path to an INI config file. It must have a section for the dataset (Section 3.3), the training setup (Section 3.3.9), the communication protocol (See section 3.3.1), the model sharing strategy (See Section 3.3.8), and the optimization method, which are options for a PyTorch optimizer like SGD. Additionally, the communication protocol section also allows for the specification of the compression algorithm (See Section 3.3.2). All this information in the INI file is parsed by The *Node* object to load the corresponding modules and classes. Additional arguments are used to initialize the classes. Hence, the behavior of the node object is highly customizable.

As several nodes might run in parallel on every machine, each node limits the number of cores that PyTorch can use. By default, PyTorch uses all the cores, which leads to significant performance degradation as the different nodes compete for the same CPU resources. Each node gets an equal fraction (the number of virtual cores divided by the number of virtual nodes) of the cores with a minimum of one core per node.

The node's main loop is in the *run* function. It trains the local model for the specified number of iterations. Each iteration starts by using the selected trainer to train the model on its dataset for the specified number of local SGD steps. After local training, the model is shared using the specified sharing method. Subsequently, many statistics, such as the number of shared bytes, are updated and written to the results file.

Another argument of the *Node* class is the number of rounds, after which the accuracy and loss are evaluated on the train and test set. In order to speed up the evaluation process, the evaluation intervals are doubled after the first twenty global epochs, quintupled after the first fifty, and decoupled after 120 global epochs.

There are two evaluation modes: local and centralized. In local evaluation, the testing and training metrics are calculated on every node using the node's current model. Hence, this mode reflects how well every node's model performs at that moment. In centralized evaluation, all nodes send their model to the node with uid zero, where they are averaged and evaluated on the test and trainset. Hence, centralized evaluation reflects how well the model would perform if, at that iteration, training would stop, and a global model would be created either by all-reduce or running distributed averaging consensus. Generally, centralized evaluation should be faster as the evaluation is run only once, and the central node is given access to all the machine's cores.

3.3.8 Sharing

The *sharing* folder contains implementations for different methods of sharing the model parameters with neighbors. All of them need to implement a *step* function, which is called in the run loop of the *Node* object.

The *Sharing* class implements D-PSGD as described in Algorithm 4. Its *step* function serializes the model into a single flattened vector, which is subsequently sent to all neighboring nodes. Afterward, it waits until it receives a model from every neighbor. Finally, the received models are deserialized and averaged with its model using the metro-hastings weights (See Equation 14).

The core functionalities of *Sharing*, such as serialization, deserialization, and averaging, are implemented in separate functions, allowing subclasses to change parts of the algorithm without reimplementing everything. Additionally, the sharing function exposes two functions that are not implemented. One is called at the beginning and the other at the end. These two functions can be used as hooks to run actions before and after sharing.

A subclass of *Sharing* is *SubSampling.* It shares a sample of the model parameters with its neighbors. A random binary mask is used to select the parameters to share. The mask can be recreated by just knowing the random seed used to create it. Hence, the seed is the only metadata making *SubSampling* a data efficient partial sharing implementation.

Another subclass of *Sharing* is *PartialModel.* It implements different versions of an importance score based on model changes. The different versions of this gauge are explained in more detail in Section 5.0.1. Some versions use both the pre and post-step hooks to update the score. In all versions, it is used to select a subset of the parameters to share with the neighbors, i.e., a partial model. Unlike *Subsampling*, there is no way to avoid sending the indices of the selected parameters. Fortunately, there are very effective algorithms to compress the list of indices, like the Elias gamma compression scheme introduced in Section 3.3.2. The *PartialModel* class is the basis for many more classes. Some of them track the score in frequency domains. Hence, a transformation method is called where necessary. By default, it is an identity function.

One of the subclasses of *PartialModel* is *Wavelet.* It uses the PyWavelets [37] implementation of DWT as the transformation function. Hence, its importance gauge decides which frequency parameters are shared. Since it shares parameters in the frequency domain, it also must average the models in the frequency domain. Therefore, it also overrides the averaging function inherited form *Sharing.*

Both *PartialModel* and *Wavelet* have bounded subclasses. These classes implement a lower bound for the number of times each parameter is shared. In each round, all parameters that were shared less than the lower bound will be subsampled to select a fixed number of them. These will then be shared along with the parameters selected based on the importance gauge. In addition, these two subclasses implement a parameter-wise averaging algorithm that is not based on metro-hastings weights. The bounded TopK algorithm is discussed in more detail in Section 5.0.5 and the parameter-wise averaging is discussed in Section 5.2.

The class *DPSGDRW* implements an alternate version of the D-PSGD algorithm. It allows for the concurrent exchange of model parameters with neighbors while the backpropagation of the model is executed. Therefore, it must be used with *TCPRandomWalk* communication class introduced in Section 3.3.1. Additionally, *DPSGDRW* supports random walk messages that are sent once a node has calculated the new average model. A completely asynchronous version of D-PSGD is implemented in *DPSGDRWAsyn*. It waits for a certain period for messages from the neighbors, and once this period expires, it uses all received models to create a new model. The period is continuously changed such that the average lag of the incoming messages is close to 0. It also does not send regular messages to the neighbors. Instead, it only sends random walk messages handled by *TCPRandomWalk*. This sharing method is used for the random walk experiments in Section 6.1. Finally, there is *SharingDynamicGraph* which is similar to Sharing but uses *TCPRandomWalkRouting*, introduced in Section 3.3.1, to get a randomly changing topology. New neighbors are added by periodically sending random walk messages that advertise the node to the final recipient of the random walk, which will establish the new connection.

3.3.9 Training

The *Training* class implements local SGD with mini-batches. It can train the model for an entire pass over the local data set or for a fixed number of mini-batches. It uses the optimizer and the loss function defined in the INI file.

3.4 Contributions

In the previous parts of this section, the DecentralizedPy framework was introduced. Some of its features were added as part of this thesis. In total, 13 pull requests were integrated into the main branch of the DecentralizPy project. The following list includes the features and changes included in these pull requests.

- The way the Femnist dataset splits the data was changed. It now splits the data among all nodes instead of just the nodes on the same machine.
- The Serialization of the data was changed from using the python JSON module to the pickle module. For synthetic vectors with five million entries, it was found that dumping vectors with pickle was more than 200 times faster than using Python's JSON library.
- An option to prevent the resetting of the optimizer after sharing was added. For optimizers such as Adam [32], resetting it results in losing its internal state.
- Each node has been limited to use $max(1, num\_cores/num\_local\_nodes)$ threads. It solved the issue of every local node using the maximum number of threads and the CPU contention it caused.
- The *PartialModel* class was reworked to track model changes. These changes were previously tracked in a separate subclass of *Training*. Moving everything into *PartialModel* allows for greater flexibility in implementing different versions of partial model sharing.
- *Sharing* was compartmentalized by moving averaging into a separate function and adding pre and post-hooks to the step function.
- *Subsampling* was implemented for partial model sharing.
- The *PartialModel* class was extended to allow tracking model changes over several communication rounds by accumulating them.
- Partial model sharing for the FFT and DWT frequency domains were added in the classes *FFt* and *Wavelet*. These subclasses of *PartialModel* share frequency domain parameters based on model change and accumulation.
- The class *TopKParams* was added. It shares the parameters with the highest absolute values.

- Support for the Reddit dataset was added. It was adopted from another project at SaCS. The RNN model for the dataset was ported to PyTorch from TensorFlow.

- The existing bash run files have been extended to support grid search and repeating experiments for different seeds. Both use the crudini application to manipulate the INI config files.

- A counting vector has been added to the sharing objects. It counts how many times every parameter has been shared. Additionally, the script *plot_percentile.py* for visualizing this information was added.

- A downscaled version of Resnet18 was added for the Femnist dataset.

- The framework was extended to work as a architecture. The newly added star graph object creates a topology with a central node acting as a parameter server. The *Synchronous* class implements model sharing between worker nodes and the parameter server.

- Added support for centralized testing to the *Node* with options to switch between centralized and local testing. Additionally, the plotting scripts were updated to support the log files of both modes.

- The plotting script *plotting_from_ csv.py* has been added. It supports replotting the plots created from *plot.py*. It principally changes the x-axis from iterations to global epochs and creates a sorted list of the results from all experiments.

- Added support for both model and metadata compression in *TCP*. Compression is enabled by a new *Compression* interface, which has several implementations for metadata and floating point compression.

These are just the contributions to the main repository. Some contributions for this thesis are kept in a private fork. These include additional implementations of *Compression* interface, the two asynchronous communication classes *TCPRandomWalk* and *TCPRandomWalkRouting* as well as the sharing strategies that make use of them, such as *DPSGDRW*, *DPSGDRWAsync*, and *SharingDynamicGraph.py*. Additionally, there are partial model versions of these sharing strategies. One final contribution is the public FastVarInt package that implements the Elias gamma compression.

4 Training and Testing Setup

4.1 Model Training

For all results presented in this thesis, the models were trained using the SGD optimization method. The model parameters were initialized with the same seed on all nodes. Using different seeds on every machine resulted in the convergence stalling at the beginning of training until the models converged enough for progress to occur. Optimization with SGD is also sensitive to the learning rate and batch sizes. Hence, a grid-search was performed for every model using the grid-search script introduced in Sections 3.2. It tunes three hyperparameters: the learning rate, the batchsize, and the number of communication rounds in a global epoch. The last hyperparameter is necessary as all tested algorithms use local SGD steps, i.e., the model is trained and updated locally between communication steps. We chose to tune the number of communication rounds per global epoch over the number of local SGD steps since the former guarantees that the number of samples used per communication round remains independent from batch sizes. Consequently, comparing the impact of the batch size for different communication budgets, i.e., communication rounds per global epoch, is easy.
An exhaustive grid search is expensive since all combinations of the three hyperparameters are tested. Therefore, we limited the scope by running it not until the convergence of the models. Additionally, before the main grid search was started, we tested a few learning rates for sensible choices of the other parameters to eliminate ones that do not converge and find a few promising ones. For every model, the hyperparameters were only tuned for D-PSGD. Therefore, the other algorithms may be disadvantaged. We also refrained from using a learning rate schedule as every algorithm may converge at different speeds, and consequently, it would be impossible to find a schedule that works well for them all. Alternatively, every algorithm could be tuned separately, but that would be prohibitively expensive.
Table 1 shows the best performing hyperparameters for every model and dataset introduced in Section 3.3.

The models were trained with the cross-entropy loss function on a 96 nodes regular graph, as explained in Section 3.3.4, of degree 4. Unless explicitly noted otherwise, all the results presented in this thesis used this graph. The nodes were run distributed across six machines, with every machine simulating 16 nodes.

4.2 Hardware Setup

All experiments were run on a combination of these machines:

- Intel(R) Xeon(R) E5-2630 v3 CPU clocked at 2.40GHz (32 virtual cores, Dual socket) with 126GB of RAM.
- Intel(R) Xeon(R) E5-2630 v3 CPU clocked at 2.40GHz (16 virtual cores) with 126GB of RAM.

Dataset	Model	lr	b	r
FEMNIST	CNN	0.01	16	10
Celeba	CNN	0.001	8	10
Reddit	RNN	0.001	16	10
CIFAR10	LeNet	0.01	8	20
Shakespeares	RNN	0.5	8	10

Table 1: Table listing the hyperparameters

4.3 Testing Setup

The results presented in this thesis can be replicated by modifying the corresponding experiment scripts to iterate over the desired list INI configuration files. These scripts were explained in more detail in Section 3.2. The main experiments were run three times with different seeds: $\{90, 91, 92\}$. These seeds impact both the model initialization as well as the data distribution.

For all experiments, node local testing was used instead of centralized testing. It was chosen because the nodes cannot perform all-reduce in practice to see how well the average model performs. Otherwise, we could use centralized training in the first place. Knowing how well the current model at every node performs is also essential if the nodes use the model while it is being trained. For example, if decentralized learning were used to train a speech recognition neural network on mobile phones, then it would be more important to know how well the model works now instead of how well it would work if we stopped training and run all-reduce. On the other hand, centralized testing would make more sense if our decentralized learning algorithm targeted training on a compute cluster. In that context, only the fully trained model matters, and all-reduce is relatively cheap to perform once at the end of training.

Node local testing is slow since every node needs to run the evaluation on the testset. Therefore, testing is not run at the end of every global epoch for the entire training duration. Instead, after 20 global epochs, testing is done every second epoch, after 50 epochs only every fifth, and after 120 epochs only every tenth.

5 JWINS++

This section introduces the JWINS++ algorithm and its core components. The algorithm consists of four parts. Part one is introduced in Section 5.0.1, and it concerns the method used to decide which parameters are shared. Several heuristics for selecting good parameters are presented. All these heuristics have one thing in common. They use the model change to decide which model parameters should be shared. Section 5.1 discusses the second part of JWINS++. It concerns how these heuristics can be used to track the model change in the wavelet frequency domain and why this might be useful. Part three is presented in Section 5.2 and has a closer look at an alternative averaging scheme to the metro-hasting averaging introduced in Section 2.3.1. Finally, the fourth part concerns the compression of the floating point parameters and is discussed in Section 5.3. These discussions of the components of JWINS++ are rounded off with a short section about the algorithm itself.

All these components and JWINS++ itself are evaluated in Section **??**. In the first part of the evaluation, experimental results are used to justify the inclusion of every component in the JWINS++ algorithm. In the second part, JWINS++ is compared against three baselines on several models.

5.0.1 Partial Model Change

Partial model sharing [27], is a well-known technique to reduce the size of shared models in decentralized learning by sharing only $100 \cdot \alpha$ percent of the model. Consequently, the amount of shared data is reduced by a factor of $\frac{1}{\alpha}$. However, this does not account for the additional metadata needed to store the indices of the shared data. This section introduces several heuristics to decide on good parameters to share.

5.0.2 TopK: Model Change

The most straightforward heuristic is to share the parameters that changed the most in every round due to training. This decision mechanism is used in Algorithm 6. It runs in parallel on all N nodes of a decentralized learning system. The algorithm uses two procedures PartialModelSharing and LocalSGD to increase readability and modularity. The first procedure shares with all neighbors the partial model of node i corresponding to the indices I_i provided as arguments. Subsequently, it receives partial models from neighbors. At this step, the node needs to wait for a partial model from all neighbors. Hence, it is effectively synchronizing them. The received partial models are then expanded with the values of the local model. Subsequently, Metro-Hastings averaging (See Section 2.3.1) is used to create a new average model. This procedure is an adapted version of the DistributedAverageConsensus procedure in Algorithm 4 for partial sharing. The second procedure of Algorithm 6 does S local SGD steps. Local SGD steps were introduced in Section 2.2.2 and are essentially just updates to the local model that are not followed by a model sharing step. In practice, we used mini-batch SGD, (See Algorithm 1) instead of SGD, which is just used to keep the description simple.

The core of the algorithm is the loop in line 18. It runs the algorithm for T communication rounds. In every round, the model change, g_i^t due to local SGD is calculated. In line 21, the indices of the model change corresponding to parameters that underwent the most change are selected. In total, 100α percent of the values are selected for sharing by this method.

5.0.3 TopK: Accumulated Model Change

The heuristic introduced in the previous section only looks at the model change of the current round. Hence, a parameter whose absolute change is always below the top 100α percent of changed parameters will never be shared. Therefore, these small changes can accumulate and lead to widely out-of-sync models at different nodes.

Algorithm 6 Partial Model Sharing with TopK

```
Require: The algorithm is run on all $N$ nodes in parallel. Every node $i \in [N]$ initializes the
    weight $x_i^{(0,0)} = x_0$. The weight matrix $W$, local dataset $D_i$, the learning rate $\eta$, the number
    of communication steps $T$, the number of local SGD steps $S$, the set of neighbors $E_i$, and
    the fraction of parameters $\alpha$ to share.
 1: procedure PartialModelSharing($x_i$, $I_i$)
 2:     send $x_i[I_i]$ to all neighbors
 3:     $x_j[I_j] \leftarrow$ receive from all neighbors for $j \neq i$  ▷ Synchronizes the node with its neighbors
 4:     $A_i \leftarrow \mathbf{0}$
 5:     for $j \in E_i$ do
 6:         $a \leftarrow x_i$
 7:         $a[I_j] \leftarrow x_j[I_j]$
 8:         $A_i \leftarrow A_i + (W)_{ij} a$
 9:     $A_i \leftarrow A_i + (W)_{ii} x_i$
10:     return $A_i$
11:
12: procedure LocalSGD($x_i^{(t,0)}$)
13:     for $s \leftarrow 0$ to $S$ do
14:         $\xi_i \leftarrow$ randomly drawn sample from $D_i$
15:         $x_i^{(t,s+1)} \leftarrow x_i^{(t,s)} - \eta \cdot \nabla f(x_i^{(t,s)}; \xi_i)$
16:     return $x_i^{(t,S)}$
17:
18: for $t \leftarrow 0$ to $T$ do
19:     $x_i^{(t+\frac{1}{2},S)} \leftarrow$ LocalSGD($x_i^{(t,0)}$)
20:     $g_i^t \leftarrow (x_i^{(t+\frac{1}{2},S)} - x_i^{(t,0)})$
21:     $I_i \leftarrow$ TopK($|g_i^t|, \alpha \cdot \mathrm{len}(x_i^{(t,0)})$)
22:     $x_i^{(t+1,0)} \leftarrow$ PartialModelSharing($x_i^{(t+\frac{1}{2},S)}, I_i$)
```

In order to curb divergence due to the accumulation of small model changes, we also accumulate in the heuristic. Hence, the Algorithm 6 is extended to accumulate the model change calculated at line 20. The model changes are accumulated into a vector v_i^t by adding the current model change g_i^t to the previous accumulation vector v_i^{t-1}. The accumulation for gradient compression (See Section 2.2.3) inspired this new heuristic. The accumulation heuristic differs from accumulation for gradient compression only so far as it does not share the accumulated model changes themselves. Instead, it is used as a score or gauge to decide which model parameters should be shared. Like in accumulation for gradient compression, the heuristic needs one more step. The accumulated changes for the shared indices need to be set back to zero. These changes are summed up in Algorithm 7. Since the algorithm does not change the two procedures PartialModelSharing and LocalSGD, they are not shown again.

Algorithm 7 Partial Model Sharing with Accumulation (v1)

```
Require: All requirements of Algorithm 6, the accumulation vector $v_i^0 = \mathbf{0}$
1: for $t \leftarrow 0$ to $T$ do
2:     $x_i^{(t+\frac{1}{2},S)} \leftarrow \text{LocalSGD}(x_i^{(t,0)})$
3:     $v_i^{t+1} \leftarrow v_i^t + (x_i^{(t+\frac{1}{2},S)} - x_i^{(t,0)})$
4:     $I_i \leftarrow \text{TopK}(|v_i^{t+1}|, \alpha \cdot \text{len}(x_i^{(t,0)}))$
5:     $v_i^t[I_i] \leftarrow \mathbf{0}$
6:     $x_i^{(t+1,0)} \leftarrow \text{PartialModelSharing}(x_i^{(t+\frac{1}{2},S)}, I_i)$
```

Algorithm 7 only accumulates the changes to the model due to local SGD. However, the model also changes due to the model averaging step. Therefore, we also account for these changes in a second version of the accumulation heuristic in Algorithm 8. As these changes are only known after the accumulation step at line 3 of Algorithm 7, two accumulation steps are needed. The second accumulation happens at the end of the loop in line 7 when the new average model is known. It adds all the changes that the model underwent, both due to the local SGD and the partial model averaging, back to the accumulation vector. We refer to this new algorithm as **TopK with Accumulation (v2)**, with the Algorithm 7 being version 1.

Algorithm 8 Partial Model Sharing with Accumulation (v2)

```
Require: All requirements of Algorithm 6, the accumulation vector $v_i^0 = \mathbf{0}$
1: for $t \leftarrow 0$ to $T$ do
2:     $x_i^{(t+\frac{1}{2},S)} \leftarrow \text{LocalSGD}(x_i^{(t,0)})$
3:     $v_i^{t+\frac{1}{2}} \leftarrow v_i^t + (x_i^{(t+\frac{1}{2},S)} - x_i^{(t,0)})$
4:     $I_i \leftarrow \text{TopK}(|v_i^{t+\frac{1}{2}}|, \alpha \cdot \text{len}(x_i^{(t,0)}))$
5:     $v_i^t[I_i] \leftarrow \mathbf{0}$
6:     $x_i^{(t+1,0)} \leftarrow \text{PartialModelSharing}(x_i^{(t+\frac{1}{2},S)}, I_i)$
7:     $v_i^{t+1} \leftarrow v_i^{t+\frac{1}{2}} + (x_i^{(t+1,0)} - x_i^{(t,0)})$
```

5.0.4 Normalized Accumulation

Normalized accumulation aims to look at model changes in relation to an individual parameter's magnitude. It does this by scaling parameter-wise the values passed to the TopK function. Hence, normalized accumulation works with both the heuristic from Section 5.0.2 as well as the accumulation heuristics in Section 5.0.3.

The idea behind using scaled model change is that a change by δv has a less relative impact on a parameter with a high absolute value compared to a parameter with a low absolute value.

Hence, small parameter undergoing a lot of relative change should be shared more often.

Normalized accumulation can be implemented by replacing line 21 in Algorithm 6 with the following line:

```
1: $I_i \leftarrow \textsc{TopK}(\frac{|g_i^t|}{x_i^{(t+\frac{1}{2},S)}+\epsilon}, \alpha \cdot \textsc{len}(x_i^{(t,0)}))$
```

For algorithms 7 and 8 it can be achieved by replacing g_i^t with v_i^{t+1} and $v_i^{t+\frac{1}{2}}$ respectively. The ϵ serves for numerical stabilization, i.e., it prevents division by zero and discounts changes on very small values that might be due to numerical round-off errors.

5.0.5 Bounded Accumulation

Bounded accumulations, like normalized accumulation, is orthogonal to the heuristics in Section 5.0.2 and 5.0.3. It sets a lower bound for the number of times each parameter must be shared. The lower bound is expressed as a fraction α_{low} of the communication budget that should be allotted to parameters shared less than the minimum bound. Hence, the minimum bound grows proportionally with the number of communications rounds. It guarantees that every parameter is expected to be shared once every $\alpha \cdot \alpha_{low}$ rounds. The bound is only upheld in expectations as from all the parameters shared less than it, only $100 \cdot \alpha_{low}$ percent of the communication budget can be used to share them. Hence, there exist parameters that are shared less than the current bound, but they will be shared eventually.

In other words, with bounded accumulation $100 \cdot \alpha$ percent of the parameters are shared every round. Of them, $100 \cdot (1 - \alpha_{low})$ percent are shared based on a model change heuristic, and $100 \cdot \alpha_{low}$ percent is shared using subsampling. The following procedure can be used to replace the TopK function in all the previously discussed model change-based partial sharing algorithms.

Algorithm 9 Bounded TopK Procedure

```
Require: counting vector $c_i = \mathbf{0}$, and lower bound fraction $\alpha_{low}$
1: procedure BoundedTopK($x_i^t, k$)
2:     $k_{TopK} \leftarrow \lceil k \cdot (1 - \alpha_{low}) \rceil$
3:     $k_{SubSample} \leftarrow k - k_{TopK}$
4:     $l_{bound} \leftarrow t \cdot \alpha \cdot \alpha_{low}$
5:     $I_{i_1} \leftarrow \textsc{TopK}(x_i^t, k_{TopK})$
6:     $c[I_{i_1}] \leftarrow c[I_{i_1}] + 1$
7:     $I_{i_2} \leftarrow \textsc{SubSample}(C[C < l_{bound}], k_{SubSample})$
8:     return concat($[I_{i_1}, I_{i_2}]$)
```

5.1 Wavelet Domain Representation

Another component of the Jwins++ algorithm is the representation of the model in the wavelet frequency domain. The transformation of the model parameters into the wavelet domain is done by using the DWT algorithm introduced in Section 2.4. With this change in the representation of the shared parameters, they no longer correspond to the changes tracked by the heuristics introduced in Section 5.0.1. Therefore, tracking and accumulating the model changes in the frequency domain is also necessary.

Algorithm 10 shows how all these changes are applied to Algorithm 8. The other two partial model sharing algorithms can be adapted similarly. The new Algorithm uses capital letters to represent variables in the frequency domain. Only changes to the core processing loop are necessary as the procedure PartialModelSharing also works for parameters in the frequency domain. The DWT is applied three times, and its inverse is calculated once.

Algorithm 10 Partial Model Sharing in the Wavelet Frequency Domain with Accumulation (v2)

```
Require: The algorithm is run on all $N$ nodes in parallel. Every node $i \in [N]$ initializes the
    weight $x_i^{(0,0)} = x_0$. The weight matrix $W$, local dataset $D_i$, the learning rate $\eta$, the number
    of communication steps $T$, the number of local SGD steps $S$, the set of neighbors $E_i$, the
    fraction of parameters $\alpha$ to share, and the accumulation vector $V_i^0 = \mathbf{0}$
1: for $t \leftarrow 0$ to $T$ do
2:     $x_i^{(t+\frac{1}{2},S)} \leftarrow \textsc{LocalSGD}(x_i^{(t,0)})$
3:     $V_i^{t+\frac{1}{2}} \leftarrow V_i^t + \mathrm{DWT}(x_i^{(t+\frac{1}{2},S)} - x_i^{(t,0)})$
4:     $I_i \leftarrow \textsc{TopK}(|V_i^{t+\frac{1}{2}}|, \alpha \cdot \textsc{len}(x_i^{(t,0)}))$
5:     $V_i^t[I_i] \leftarrow \mathbf{0}$
6:     $Z_i^{(t+\frac{1}{2},S)} \leftarrow \mathrm{DWT}(x_i^{(t+\frac{1}{2},S)})$
7:     $Z_i^{(t+1,0)} \leftarrow \textsc{PartialModelSharing}(Z_i^{(t+\frac{1}{2},S)}, I_i)$
8:     $x_i^{(t+1,0)} \leftarrow \mathrm{DWT}^{-1}(Z_i^{(t+1,0)})$
9:     $v_i^{t+1} \leftarrow v_i^{t+\frac{1}{2}} + \mathrm{DWT}(x_i^{(t+1,0)} - x_i^{(t,0)})$
```

Applying the wavelet transformation to the model parameters has two advantages. Firstly, The transformation separates the model vector into a low and high-frequency representation. Secondly, it decomposes the signal into several levels. Thus, sharing a frequency parameter from a high level will provide information about changes that occurred to an area around the parameter.

5.2 Parameter-wise Averaging

The partial sharing algorithms introduced in Section 5.0.1 works by sharing a fraction of the parameters $x \in \mathbb{R}^n$ with all the neighbors. In a communication graph is $G = (V, E)$ these neighbors of a node V_i with $i \in [N]$ can be expressed as $M_i = \{j : (j, i) \in E\}$. As every node shares a different subset of the parameters with its neighbors, it also receives a different subset from every neighbor. However, we use metro-hastings weights the same way as if we shared the entire model like in D-PSGD. A look at the actual communication graph $G_k^t = (V, E^t)$ for parameter $(x^t)_k$ at communication round t reveals that a node V_i may send its parameter $(x_i^t)_k$ to a neighboring node V_j with $j \in M_i$, which means $(i, j) \in E^t$, but the reverse might not be true, i.e., V_j does not send $(x_j^t)_k$ to V_i and so $(j, i) \notin E^t$. Therefore, partial model averaging does not apply metro-hastings averaging on the level of parameters. The sharing graph is not even symmetric but directed instead. Hence, convergence to the global average of parameter $(x)_j$ is not guaranteed by using the consensus-averaging algorithm 2.3.1. Fortunately, if we can guarantee consecutive periods of length τ for which the union of all communication graphs is connected, then distributed consensus-averaging will converge [29], which is the case when bounded accumulation is used. However, it does not necessarily converge to the global average. Therefore, it makes sense to look if alternatives to the metro-hastings weights work better. Other algorithms in Decentralized Learning average the parameters [29], with others applying correction methods to guarantee convergence to the global average [48, 2]. For partial model sharing, parameter-wise averaging [27] has also been used to merge the received sample with the local weights. Hence, this section aims to analyze the impact of parameter-wise averaging on the convergence speed of consensus-averaging. An important metric for the convergence speed of distributed consensus-averaging is the eigengap of the weight matrix. Since our effective weight matrix is different for every parameter, we will analyze the expected weight matrix for random subsampling with factor α. In this section, W_k^t will denote the expected weight matrix for parameter k at time t, while W denotes the metro-hastings matrix corresponding to 100% sharing. In the partial model implementation of D-PSGD (See Algorithm 6), if at node i no values are received from neighbor j for parameter x_k at round t, then they are substituted with the local values from node i. This substitution is equivalent to increasing the weight of $(W_k^t)_{ii}$ by the weight $(W)_{ik}$. Hence the effective weight matrix W_k^t for parameter k at round t is 0 for all neighbors that did not send a value. Therefore, the expected weight matrix for parameter k at time t corresponds to:

$$(W_k^t)_{ij} = \begin{cases} \alpha \cdot (W)_{ij} & (j,i) \in E \land j \neq i \\ 1 - \sum_{j \in M_i} \alpha \cdot (W)_{ij} & j = i \\ 0 & else \end{cases} \tag{19}$$

The equation for the self-weight can be reformulated to make the representation of the matrix simpler:

$$(W_k^t)_{ii} = 1 - \sum_{j \in M_i} \alpha \cdot (W)_{ij} = \alpha - \sum_{j \in M_i} \alpha \cdot (W)_{ij} + (1 - \alpha) \tag{20}$$

Therefore, the expected weight matrix for all parameters k at any round t can now be easily represented in matrix form:

$$(W_k^t) = \alpha \cdot W + (1 - \alpha) \cdot I \tag{21}$$

From the matrix form of Equation 21 one can infer how the eigenvalues λ_i' of W_k^t can be expressed with the eigenvalues λ_i of the weight matrix W:

$$\lambda_i' = 1 + \alpha(\lambda_i - 1) \tag{22}$$

As described in section 2.3.1, the convergence rate of distributed consensus averaging depends on the eigengap, which is the difference between the largest and second largest absolute eigenvalues. The eigenvalues of W lie between 1 and -1 with 1 being the highest one, i.e., $1 \geq \lambda_2 \geq ... \geq \lambda_n \geq -1$. This is the case as it fulfills the Equations 9 to 11. Since W_j^t can be expressed in terms of W, and the eigenvalues of the latter get transformed by Equation 22. This means W_j^t lie also between -1 and 1 and 1 is still the highest eigenvalue. Hence, only the second largest eigenvalue changes. Let $\lambda_{max}' = \max(|\lambda_2'|, |\lambda_n'|)|$ then

$$\rho' = 1 - \lambda_{max}' = 1 - 1 + \alpha - \alpha|\lambda_{\{2,n\}}| = \alpha(1 - \lambda_{\{2,n\}}) \geq \alpha \cdot \rho \tag{23}$$

with $\lambda_{\{2,n\}}$ being either λ_2 or λ_n. As the transformation of the eigenvalues by Equation 22 might change which one is larger, the new eigengap can only be lower bounded.

For a regular graph with degree d (See Section 3.3.4) we can do better. Each node can simply parameter-wise average the incoming vectors, e.g., node i receives two values for parameter $(x)_k$ from neighbors $l, j \in M_i$ then it averages them $(x_i)_j^{t+1} = ((x_i)_k^t + (x_l)_k^t + (x_j)_k^t)/3$. For d neighbors a node will receive on average $\alpha \cdot d$ values per parameter. Hence in expectation the average is taken over $\alpha \cdot d + 1$ parameters. Therefore the self weight is:

$$(W_k^t)_{ii} = \frac{1}{\alpha \cdot d + 1} \tag{24}$$

The weights of all neighbors $j \in M_i$ must be equal as there is the same chance for every neighbor to send the parameter x_k. Therefore $1 - (W_k^t)_{ii}$ gets equally distributed among them:

$$(1 - \frac{1}{\alpha \cdot d + 1}) \cdot \frac{1}{d} = \frac{\alpha \cdot d^2}{\alpha \cdot d^3 + d^2} = \frac{\alpha}{\alpha \cdot d + 1} \tag{25}$$

So, putting it all together it can be expressed with the following weight matrix:

$$(W_k^t)_{ij} = \begin{cases} \frac{\alpha}{\alpha \cdot d + 1} & (j,i) \in E \land j \neq i \\ \frac{1}{\alpha \cdot d + 1} & j = i \\ 0 & else \end{cases} \tag{26}$$

Expressing the above equation with respect to W_j, the metro-hasting matrix of the regular graph, yields:

$$(W_k^t)_{ij} = \frac{\alpha \cdot (d+1)}{\alpha \cdot d + 1} \cdot W + \frac{1-\alpha}{\alpha \cdot d + 1} \cdot I \tag{27}$$

Therefore, the eigenvalues for the new averaging scheme are:

$$\lambda_i' = \frac{\alpha \cdot (d+1)}{\alpha \cdot d + 1} \cdot \lambda_i + \frac{1-\alpha}{\alpha \cdot d + 1} \tag{28}$$

To figure out the impact on the eigengap, we will use $\lambda'_{max} = \max(|\lambda'_2|, |-\lambda'_n|)|$

$$\rho' = 1 - \lambda'_{max} = 1 - \frac{\alpha \cdot (d+1)}{\alpha \cdot d + 1} \cdot |\lambda_{\{2,n\}}| - \frac{1-\alpha}{\alpha \cdot d + 1} \geq \frac{\alpha \cdot (d+1)}{\alpha \cdot d + 1} \cdot \rho \tag{29}$$

Finally, the two eigengaps can be compared:

$$\frac{\alpha \cdot (d+1)}{\alpha \cdot d + 1} \cdot \rho \cdot \frac{1}{\alpha \cdot \rho} = \frac{d+1}{\alpha d + 1} \tag{30}$$

For $\alpha = 0.1$ and a degree of $d = 4$ the parameter-wise averaging scheme has a 3.57 times larger eigengap, which translates to faster convergence speeds according to Equation 13

5.3 Lossy Compression

Lossy compression on the floating-point parameters of the model reduces their size at the cost of introducing reconstruction error. The idea behind using lossy compression is that neural networks have been shown to work at low-bit representations [24, 18]. Hence, model averaging should be able to handle some of the errors induced by compression.

If lossy compression with compression factor r is used in conjunction with partial model sharing, then the total exchanged data is reduced by a factor of $\frac{1}{alpha} \cdot r$.
For jwins++, four lossy compression techniques were considered. These are 16-bit floating-point numbers, 8-bit quantization, fpzip [42] compression, and zfp [41] compression.

16-bit floating point representation. 16-bit floating-point numbers are natively supported in PyTorch. Using 16-bit floats over the default 32bit floating-point numbers yields a compression factor of 2.

8-bit Quantization. 8-bit quantization is another model compression mechanism natively supported by PyTorch. It represents every parameter with only 8 bits. Hence, its compression factor is four, not counting the negligible metadata of one floating point number and a byte. The floating point number represents the scale s and the byte the zero point z_0 of the quantization. They are needed to convert the quantized parameters back to floating point numbers. The reconstruction works like this: For a quantized parameter with the value q, the reconstructed floating point value is $(q - z_0) \cdot s$.

There are different options for how the scale and zero point are chosen. We used affine min-max quantization for each vector v. It sets the scale to $(v_{max} - v_{min})/256$ and the zero point is set to $\text{ROUND}(v_{min}/s)$.

fpzip. Fpzip is a floating point compression algorithm that offers both lossy and lossless compression for 32-bit and 64-bit floating-point numbers. In lossy mode, it reduces the size of the mantissa during the compression process. The size of the exponent cannot be reduced, limiting its usefulness for vectors of predominately small values like most of our models.

zfp. Zfp is another lossy floating point number compression algorithm. It is designed to provide high throughput and offers the option to set a bound for the compression error.

5.4 The Jwins++ Algorithm

The Jwins++ algorithm is a continuation of the Jwins algorithm [53]. It removes the randomized cut-off scheme that randomly selects the communication budget for each round from a predefined list of values. It is replaced with the bounded accumulation algorithm introduced in section 5.0.5. The bounded accumulation makes sure that all parameters get shared eventually. Therefore, it removes the main reason randomized cut-offs were introduced. An additional advantage of the bounded accumulation scheme is that one can change the communication by manipulating a single hyperparameter α. Contrary to randomized cut-offs, for which one would need to define a new list of alphas. An additional departure from Jwins is the use of parameter-wise averaging, introduced in Section 5.2. It aims to speed up convergence, especially for smaller communication budgets α. In addition to compression through subsampling, Jwins++ further reduces its data size by applying zfp compression with a tolerance of 0.001.

However, despite these changes, the core part of the algorithm remains the same. Both Jwins++ and Jwins use the accumulated model change importance gauge (Algorithm 8) to decide which parameters to share and use the Elias gamma algorithm to reduce the size of the metadata. Additionally, both algorithms track these changes in the wavelet frequency domain of the Sym2 wavelet. One last notable difference is that in Jwins, the decomposition was only done for four levels, while Jwins++ does a complete decomposition.

The Jwins++ algorithm is described in Algorithm 11. It shows the entire algorithm except for the Elias gamma compression of the index, as the sending of the index is implicitly handled by the pseudocode in lines 2 and 3. One yet unseen change is that the PartialModelSharing procedure has been changed to do parameter-wise averaging.

5.5 Evaluation

This section evaluates the Jwins++ algorithm as well as its constituent parts. The results for the components of Jwins++ are evaluated first. Their evaluation starts with the model change heuristics that includes the TopK heuristic of Section 5.0.2, the accumulation heuristics of Section 5.0.3, as well as two sections for normalized and bounded accumulation of Sections 5.0.5 and 5.0.4. These sections are followed by a section that evaluates the wavelet frequency representation introduced in Section 5.1 and a section on the parameter-wise averaging scheme introduced in Section 5.2. Finally, several algorithms for model compression, as introduced in Section 5.3, are evaluated. Each section's results experimentally justify the component's inclusion or omission from the Jwins++ algorithm. The results were obtained by training the CNN model for Femnist over 120 epochs.
The second part of this section evaluates the Jwins++ algorithm on several models and datasets. For each of them, the Jwins++ algorithm is compared against three baselines: 100% sharing, subsampling, and accumulation. The results are all rerun three times and evaluated for both 10% and 30% partial model sharing. More information about how the models were trained and evaluated can be found in Section 4.
All results are presented with graphs depicting the training and testing loss, the testing accuracy, and a supplementing table that shows the highest average accuracy and its standard deviation. The graphs consist of a line plot with error bands. Since node local training was used, the evaluation happened on every node. Hence, the line plot represents the average value of the metric across all machines. For experiments run more than once, it represents the average of the averages over all runs. The error bands depict the standard deviation of these node local results.

Algorithm 11 JWINS++

```
Require: The algorithm is run on all $N$ nodes in parallel. On every node $i \in [N]$ initializes the weight $x_i^{(0,0)} = x_0$. The local dataset $D_i$, the learning rate $\eta$, the number of communication steps $T$, the number of local SGD steps $S$, the set of neighbors $E_i$, the fraction of parameters $\alpha$ to share, the accumulation vector $V_i^0 = \mathbf{0}$, counting vector $c_i = \mathbf{0}$, and lower bound fraction $\alpha_{low}$
 1: procedure PartialModelSharing($x_i$, $I_i$)
 2:     send $x_i[I_i]$ to all neighbors
 3:     $x_j[I_j] \leftarrow$ receive from all neighbors for $j \neq i$    ▷ Synchronizes the node with its neighbors
 4:     $A_i \leftarrow \mathbf{0}$
 5:     $C_i \leftarrow \mathbf{0}$
 6:     for $j \in E_i$ do
 7:         $a \leftarrow x_i$
 8:         $a[I_j] \leftarrow x_j[I_j]$
 9:         $C[I_j] \leftarrow C[I_j] + 1$
10:         $A_i \leftarrow A_i + a$
11:     $A_i \leftarrow A_i + x_i$
12:     $A_i \leftarrow A_i / C_i$
13:     return $A_i$
14: procedure LocalSGD($x_i^{(t,0)}$)
15:     for $s \leftarrow 0$ to $S$ do
16:         $\xi_i \leftarrow$ randomly drawn sample from $D_i$
17:         $x_i^{(t,s+1)} \leftarrow x_i^{(t,s)} - \eta \cdot \nabla f(x_i^{(t,s)}; \xi_i)$
18:     return $x_i^{(t,S)}$
19: procedure BoundedTopK($x_i^t, k$)
20:     $k_{TopK} \leftarrow \lceil k \cdot (1 - \alpha_{low}) \rceil$
21:     $k_{SubSample} \leftarrow k - k_{TopK}$
22:     $l_{bound} \leftarrow t \cdot \alpha \cdot \alpha_{low}$
23:     $I_{i_1} \leftarrow$ TopK($x_i^t, k_{TopK}$)
24:     $c[I_{i_1}] \leftarrow c[I_{i_1}] + 1$
25:     $I_{i_2} \leftarrow$ SubSample($C[C < l_{bound}], k_{SubSample}$)
26:     return concat($[I_{i_1}, I_{i_2}]$)
27: for $t \leftarrow 0$ to $T$ do
28:     $x_i^{(t+\frac{1}{2},S)} \leftarrow$ LocalSGD($x_i^{(t,0)}$)
29:     $V_i^{t+\frac{1}{2}} \leftarrow V_i^t + \mathrm{DWT}(x_i^{(t+\frac{1}{2},S)} - x_i^{(t,0)})$
30:     $I_i \leftarrow$ BoundedTopK($|V_i^{t+\frac{1}{2}}|, \alpha \cdot \mathrm{len}(x_i^{(t,0)})$)
31:     $V_i^t[I_i] \leftarrow \mathbf{0}$
32:     $Z_i^{(t+\frac{1}{2},S)} \leftarrow \mathrm{DWT}(x_i^{(t+\frac{1}{2},S)})$
33:     $Z_i^{(t+1,0)} \leftarrow$ PartialModelSharing($Z_i^{(t+\frac{1}{2},S)}, I_i$)
34:     $x_i^{(t+1,0)} \leftarrow \mathrm{DWT}^{-1}(Z_i^{(t+1,0)})$
35:     $v_i^{t+1} \leftarrow v_i^{t+\frac{1}{2}} + \mathrm{DWT}(x_i^{(t+1,0)} - x_i^{(t,0)})$
```

5.5.1 TopK: Model Change

This section evaluates the heuristic introduced in Section 5.0.2 that shares the TopK most changed parameters due to the local SGD steps. It is compared to 100% sharing, i.e., D-PSGD, random subsampling that randomly chooses the parameters to share, and TopK parameters, which shares the highest absolute valued parameters. All three partial sharing variants used an α of 0.1, i.e., they shared 10% of the model in every communication round. The comparison is shown in Figure 2 and the best accuracies are listed in Table 2

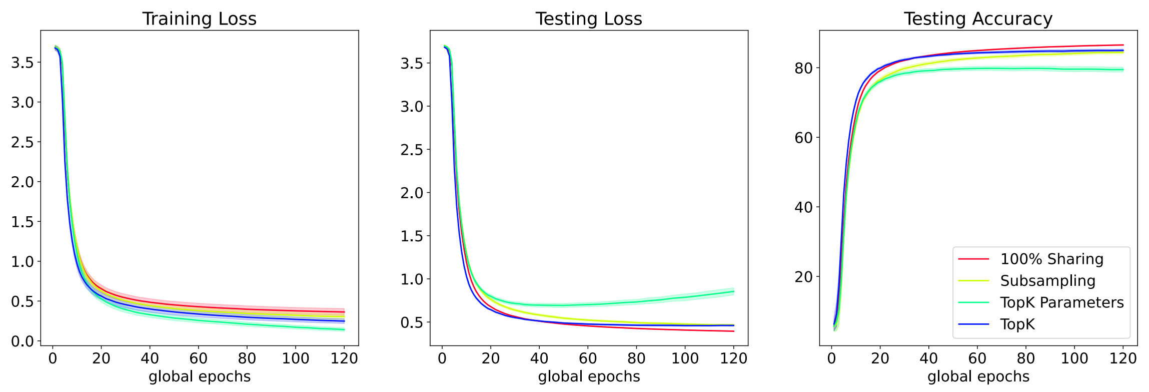

Figure 2: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Three partial sharing methods (TopK, TopK Parameters, Subsampling) are compared against 100% sharing.

Algorithm	Avg	Std
100% Sharing	86.549	0.084
TopK Model Change,	84.991	0.368
Subsampling	84.407	0.423
TopK Parameters	79.841	0.606

Table 2: Highest accuracies and their standard deviations

The highest accuracy was achieved for 100% sharing followed by a wide margin by TopK model change. It is closely followed by subsampling. Lastly, TopK parameter is the distant last. Its accuracy stopped improving, and its testing loss has already begun to diverge. Its problems are shown in Figure 3, which shows how many times each parameter is shared. For TopK parameters, 80 percent of parameters are never shared. Hence, the changes undergone by these parameters are not propagated, and the models begin to overfit to the local data. The overfitting to the local data might also explain why it achieved the lowest training loss, as training loss is measured on the local training data.
Another interesting conclusion that can be drawn from these plots is that subsampling converges slower than 100% sharing and TopK model change. A possible explanation is that it takes until the important parameters are randomly chosen to be shared, but because they are eventually shared, it will catch up to the others. Hence, by the end, it has almost caught up with TopK model change.

5.5.2 TopK: Accumulated Model Change

In this section, two implementations of the TopK accumulated model change heuristic are compared against each other, 100% sharing and the TopK model change heuristic. These two versions of accumulation were introduced in 5.0.3. The first version only accumulates changes due to training-induced model changes. The other version also accounts for changes due to the

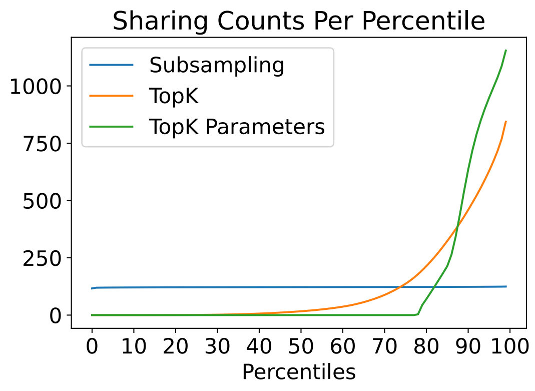

Figure 3: Average share counts for the parameters sorted into percentiles for the methods TopK, TopK Parameters, Subsampling

averaging step.
As in the previous section, the partial model algorithms were evaluated for an α of 0.1. The plots that compare these models are shown in Figure 4 and table 3 lists the achieved accuracies.

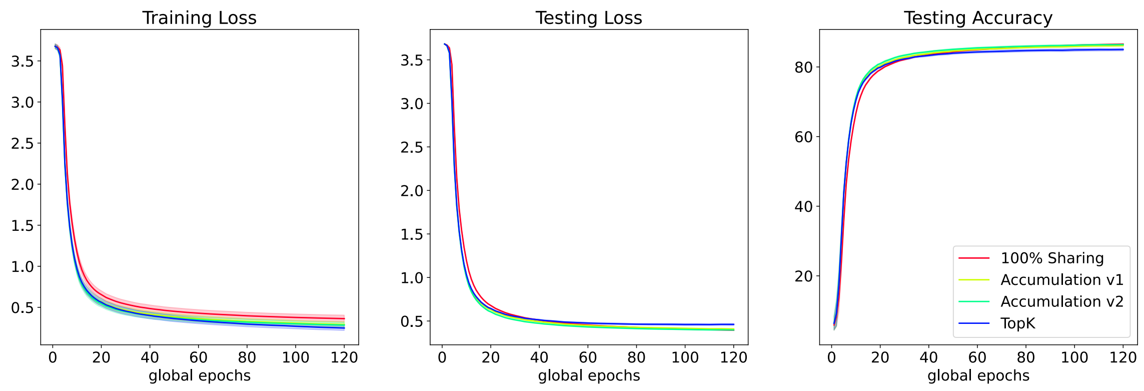

Figure 4: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Three partial sharing methods (TopK, TopK Accumulation v1-2) are compared against 100% sharing.

Algorithm	Avg	Std
100% Sharing	86.549	0.084
Accumulation v2	86.410	0.238
Accumulation v1	86.119	0.240
TopK Model Change	84.990	0.368

Table 3: Highest accuracies and their standard deviations

All two versions of accumulation outperformed TopK model change for the measured accuracies and the test loss. However, they did not reach the accuracy achieved by 100% Sharing. The second version of accumulation outperformed the other one for the test accuracy and loss and almost reached the same results as 100% sharing. Noticeable is that all partial sharing variants initially outperform the 100% sharing baseline. However, after the first 30 epochs, full sharing begins to catch up and surpasses them by the end.
All three partial sharing methods have lower training loss, indicating that the partial sharing models overfit slightly to the local data. One reason for the overfitting may be that even for accumulation, most parameters are rarely shared, as seen in Figure 5. Hence, these parameters

are optimized almost exclusively for the local dataset. However, the plot also shows that both versions of accumulation are more effective at sharing slowly changing parameters than TopK model change. So far, the results suggest that sharing the most frequently changing parameters speeds up convergence in the beginning, but neglecting slowly changing parameters hurts the convergence performance in the long run. Therefore, a balance between these two needs to be struck.

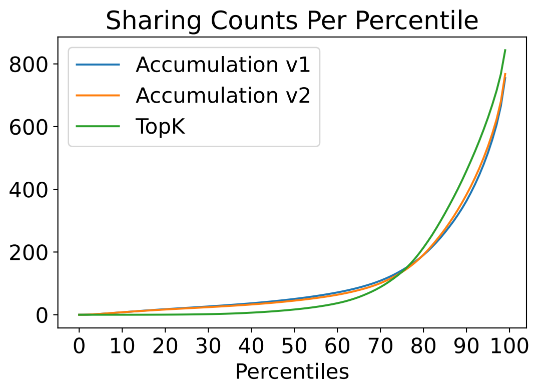

Figure 5: Average share counts for the parameters sorted into percentiles for the methods TopK, TopK Accumulations v1-2

Suppose overfitting to the local dataset is a problem. In that case, increasing the factor α should yield higher accuracies towards the end of the training, and this is exactly what an evaluation of the Accumulation v2 algorithms for different α showed. The results are shown in Figure 6 and Table 4. The worst results are achieved if only 1% of the data is shared in every round, closely followed by sharing 5%. The differences between sharing 10%, 20%, and 30% are much smaller. With the latter two essentially reaching the same final accuracy.

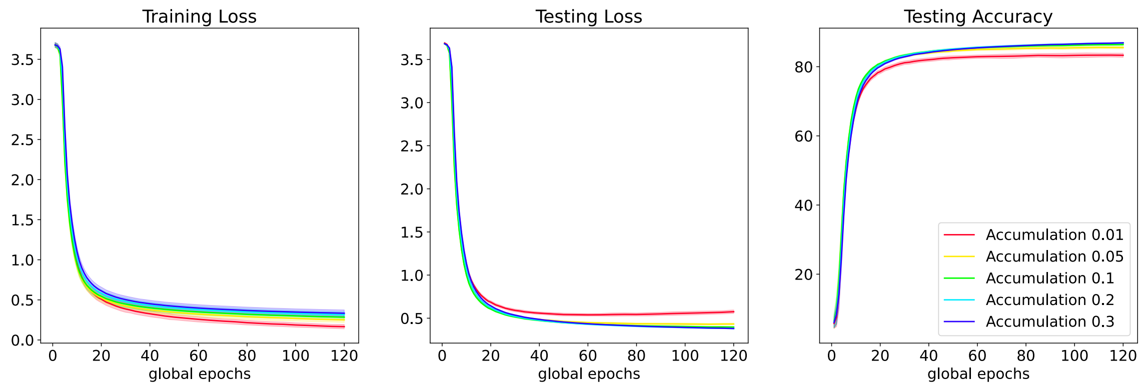

Figure 6: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. It shows the results for Accumulation 2 for different α.

Algorithm	Avg	Std
Accumulation 0.3	86.889	0.0979
Accumulation 0.2	86.887	0.127
Accumulation 0.05	85.576	0.367
Accumulation 0.01	83.351	0.537

Table 4: Highest accuracies and their standard deviations

5.5.3 Normalized Accumulation

The results for normalized accumulation were evaluated using the normalized TopK algorithm outlined in Section 5.0.4 on top of the Accumulation v2 algorithm. Two numerical stability factors ϵ 0.01 and 0.0001 were tested and compared against accumulation. The results are shown in Figure 7 and the top accuracies are listed in Table 5.

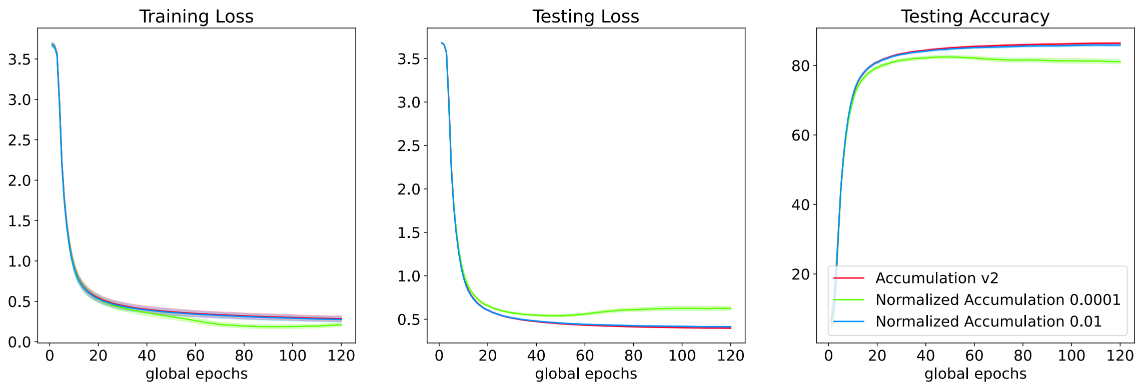

Figure 7: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Two versions of normalized accumulation are compared against regular accumulation.

Algorithm	Avg	Std
Accumulation v2	86.410	0.238
Normalized Accumulation 0.01	85.875	0.320
Normalized Accumulation 0.0001	82.446	0.479

Table 5: Highest accuracies and their standard deviations

The results show that normalized TopK does not work better than regular TopK. Especially for the small numerical stability factor, the model begins to diverge. Hence, we can conclude that looking at the relative change of the parameters does not work. For the larger numerical stability factor, better results were achieved. However, they are still worse than using regular TopK accumulation, and using an even larger stability factor does not make sense as almost all gradient values are relatively small.

5.5.4 Bounded Accumulation

The experiments of this section compare two versions of bounded accumulation (See Section 5.0.5) against the accumulation v2 algorithm. Both bounded versions are built on top of accumulation v2 and differ only in their values for α_{low}. For it, two values were chosen 0.1 and 0.2, which means that 10% and 20% of the communication budget were allotted to parameters that were shared less than $100\alpha \cdot \alpha_{low}$ percent of the time.

All three algorithms were evaluated for an α of 0.1 over three different runs. The results are shown in Figure 8 and the top accuracies are listed in Table 6.

Algorithm	Avg	Std
Bounded Accumulation 20%	86.451	0.23
Bounded Accumulation 10%	86.437	0.205
Accumulation v2	86.395	0.233

Table 6: Highest average accuracies and their standard deviations over three runs

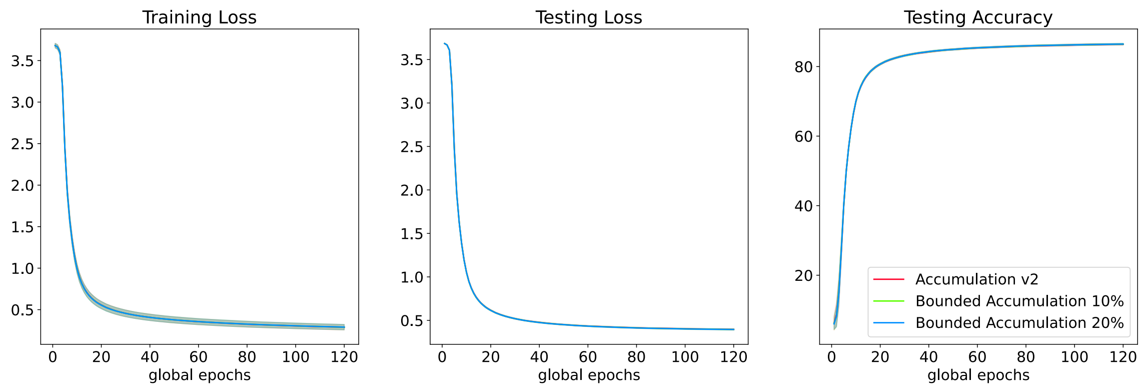

Figure 8: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Two versions of bounded accumulation are compared against regular accumulation.

All three algorithms are very competitive. The plots show that there is essentially no difference in the convergence speed between them. However, across all three runs, both bounded accumulation algorithms beat the accumulation baseline in terms of accuracy. The best performing algorithm was bounded accumulation with a 20% lower bound. These results support the hypothesis that neglecting the slowly changing parameters hurts convergence as bounded accumulation, which allots bandwidth to share them, outperformed regular accumulation. The impact of bounded accumulation on the shared parameters can be seen in Figure 9. The impact is small as bounded accumulation only guarantees that parameters are shared once every $\frac{1}{\alpha \cdot \alpha_{low}}$ communication rounds, which for the tested lower bounds of 10% and 20% yields 100 and 50 rounds respectively.

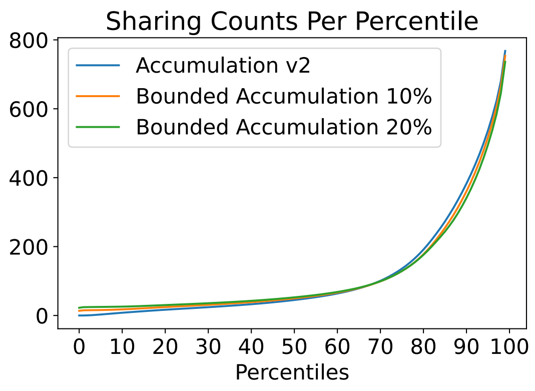

Figure 9: Average share counts for the parameters sorted into percentiles for the methods TopK Accumulations v2, and Bounded Accumulation 10% and 20%

5.5.5 Wavelets

This section evaluates the accumulation of the model changes in the wavelet domain as described in Section 5.1. The second version of the accumulation algorithm was used for the accumulation, and the α was set to 0.1.

The transformation of the model values into the wavelet domain was done via the DWT described in Section2.4. It requires two hyperparameters, the wavelet function and the number of levels for which the wavelet decomposition is done. In Table 7, the results for several hyperparameter combinations are listed.

Wavelet	Levels	Avg	Std
Rbio22	Max	86.494	0.215
Rbio44	Max	86.487	0.218
Sym2	Max	86.475	0.217
Bior44	Max	86.471	0.220
Sym7	Max	86.453	0.225
Bior22	Max	86.429	0.219
Db7	Max	86.390	0.234
Rbio22	4	86.388	0.236
Rbio44	4	86.386	0.249
Sym2	4	86.374	0.239
Bior44	4	86.374	0.243
Sym7	4	86.356	0.256
Bior22	4	86.329	0.243
Db7	4	86.2951	0.263

Table 7: Highest accuracies achieved for the model change accumulation in the wavelet domain using the listed wavelet function and decomposition levels. Max means that Pywt [37] chose the maximal possible level for the given input vector

For each wavelet function, better results were achieved when the maximal decomposition level was chosen. Hence, it is the decomposition into lower-level representations of the model vector that makes the accumulation in the wavelet domain outperform accumulation in the model's spatial domain and not the splitting of the vector into a low and high-frequency representation.

While the gains of using wavelet are small for the CNN model on the Femnist dataset (86.410% accuracy vs. 86.494%), they are much more substantial for LeNet on Cifar10 as shown in the JWINS paper [53].

As the results in table 7 are all close to each other, the four best-performing wavelet functions were rerun two more times to give average accuracies over three runs. The results are listed in Table 8. All of them are still very close to each other, but the order shifted slightly. Now the Sym2 wavelet is the best performing one. Therefore, the Sym2 wavelet with the maximal decomposition level will be used in all wavelet accumulation experiments.

Wavelet	Levels	Avg	Std
Sym2	Max	86.451	0.204
Rbio22	Max	86.446	0.214
Bior44	Max	86.441	0.216
Rbio44	Max	86.440	0.213

Table 8: Highest accuracies achieved for the model change accumulation in the wavelet domain using the listed wavelet function and decomposition levels. The average results over three runs are reported.

5.5.6 Parameter-wise Averaging

In this section, the averaging scheme introduced in Section 5.2 is compared to distributed consensus averaging with metro-hastings weights (See Section 5.2). Both averaging schemes are used once with bounded accumulation (v2) and once with bounded accumulation in the sym2 wavelet domain. In both cases, the bounded accumulation used the hyperparameter α was set to 0.1 and the lower bound α_{low} to 0.2.

The parameter-wise averaging scheme achieves higher convergence results in both the model spatial and wavelet domains. Besides reaching higher accuracies, it also converges faster, which

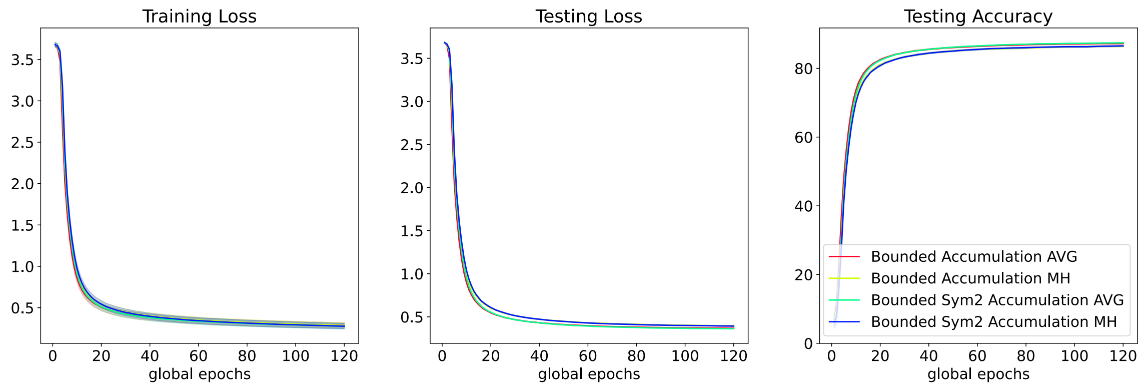

Figure 10: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Two bounded accumulation implementations are used to compare metro-hastings (MH) against parameter-wise averaging (AVG).

Algorithm	Avg	Std
Bounded Sym2 Accumulation AVG	87.429	0.152
Bounded Accumulation AVG	87.223	0.185
Bounded Sym2 Accumulation MH	86.521	0.186
Bounded Accumulation MH	86.417	0.260

Table 9: Highest average accuracies and their standard deviations

can be seen in the testing accuracy plot and loss plot. The two parameter-wise averaging schemes pull ahead from the beginning and keep their lead until the end.

For both metro-hastings and parameter-wise averaging, the bounded accumulation in the wavelet domain achieved higher accuracies. One possible explanation could be given by Figure 11. It shows that the wavelet versions share the 20% most shared parameters slightly less in absolute terms, which is compensated by sharing the parameters between the 50th and 80th percentile more.

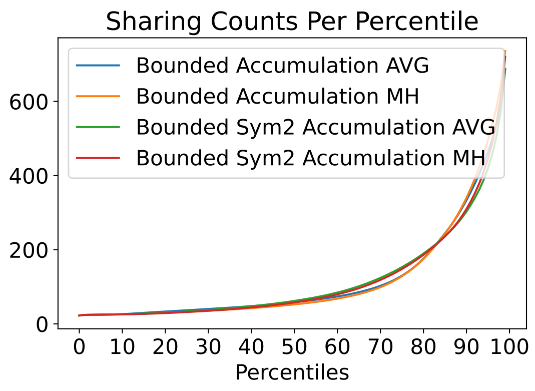

Figure 11: Average share counts for the parameters sorted into percentiles for the methods bounded accumulaiton in spacial and wavelet domain as well as with metro-hastings (MH) and parameter-wise averaging (AVG).

5.5.7 Lossy Model Compression

This section applies the model compression schemes discussed in Section 3.3.2 to the 10% of the data shared by the accumulation v2 algorithm in the wavelet domain, which also serves as the baseline. Of the four discussed compression schemes, only three converged: 16-bit floats, fzip, and zfp compression. For the latter two, two compression strengths were tested. The 8-bit quantization did not converge for values in the wavelet domain. Therefore, it will not be shown in this section. The results of these five tests are plotted in Figure 12, and their highest accuracies are listed in Table 10.

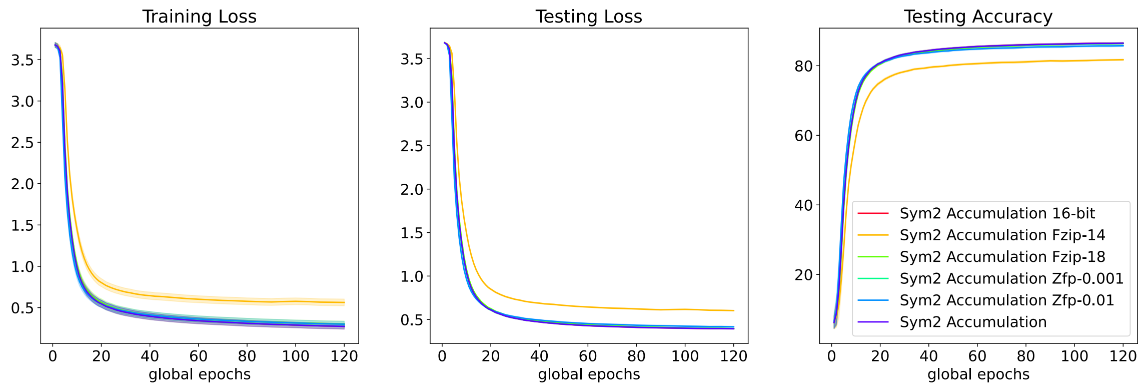

Figure 12: Training loss, testing loss, and testing accuracy for 120 epochs of the Femnist CNN model. Five different compression schemes are compared.

Algorithm	Avg	Std
Sym2 Accumulation 16-bit	86.478	0.213
Sym2 Accumulation	86.475	0.217
Sym2 Accumulation Fzip-18	86.420	0.1971
Sym2 Accumulation Zfp-0.001	86.419	0.215
Sym2 Accumulation Zfp-0.01	85.751	0.240
Sym2 Accumulation Fzip-14	81.727	0.209

Table 10: Highest average accuracies and their standard deviations

Surprisingly, the best results were not achieved for the baseline but for the 16-bit floating point representation of the model data. The Fzip algorithm achieved the third best result with the precision set to 18. However, it compresses the model data only by 2.04x. Hence, its use makes little sense compared to the 16-bit floating point representation. Fzip is almost tied by zfp compression with the error bound set at 0.001, and the latter has a much higher compression ratio of 3.09x. For the last two algorithms, the compression significantly impacted the achieved accuracies. Fpzip with an error bound of 0.01 compresses the data by 4.38x, but it loses more than 0.6 accuracy percentage points. The worst performance was achieved with fzip with precision set to 14, which has a compression ratio of 2.74x. In all three plots its results are worse from almost the beginning. In Figure 13, the total amount of data exchanged with a neighbor is shown for all six algorithms. In addition to floating point compression, the metadata was compressed with the Elias gamma algorithm (See Section 3.3.2) by 6.78x.

The algorithm that offers the best compression compared to its achieved accuracy is zfp with the 0.001 error bound. Hence, it is the compression algorithm used in Jwins++. It, together with Elias gamma, compresses all the data, i.e., the model parameter and the index metadata, by a factor of 4.23x.

5.5.8 Jwins++

The main section of this thesis is concluded by an evaluation of the Jwins++ algorithm on the following five datasets: Femnist, Celeba, Reddit, Shakespeares, and Cifar10. For each dataset,

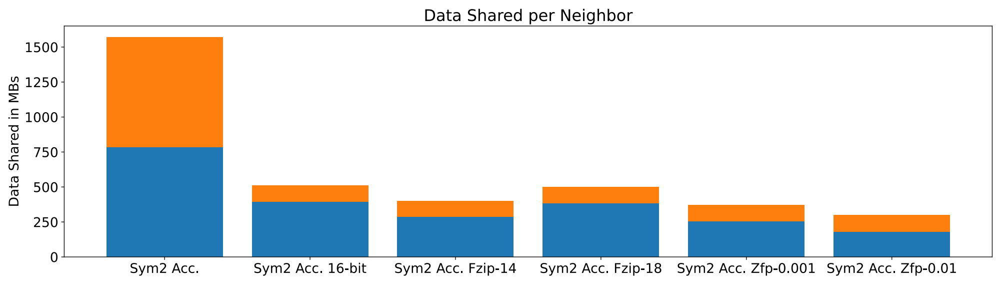

Figure 13: Amount of data shared with all the compression schemes

the JWINS++ algorithm is compared against two other partial model sharing algorithms and D-PSGD, i.e., full sharing. The two partial sharing algorithms are random subsampling, and accumulation v2 (See Algorithm 8). All the partial model algorithms are tested for an α of 0.1 and 0.3. Hence, they share the same number of parameters in every round. However, the total amount of transmitted bytes will vary as subsampling and accumulation are run without lossy model compression. Additionally, JWINS++ and the accumulation algorithm require sending an index along with the parameters, which is, despite the effective compression with the Elias gamma algorithm (See Section 3.3.2), still a substantial overhead compared to subsampling.

The JWINS++ algorithm is run with a lower bound α_{low} of 0.2, the Sym2 wavelet as the basis function for the DWT that is run for the maximal number of decomposition steps. Other hyperparameters, such as the learning rate, batch size, and communication rounds per epoch, are the same for all four algorithms. However, they are different for each dataset. A list of all the hyperparameter choices is given in Table 1.

Femnist. In the previous sections, the core components were evaluated on Femnist over 120 global epochs. For this final evaluation, the CNN model was trained on Femnist for 240 epochs. The results for 10% sharing are shown in Figure 14, the ones for 30% sharing are shown in Figure 15, and the final accuracies for the algorithms are given in Table 11.

For 10% sharing, the JWINS++ algorithm beats all others, even 100% sharing. It noticeably pulls ahead of all other algorithms from the beginning in both the testing loss and accuracy plots. It also beats the TOPK accumulation baseline by almost 0.8 percentage points and the subsampling baseline by more than 2.2 points.

For 30% sharing, the JWINS++ algorithm only improved by a bit more than 0.1 and is even beaten by the accumulation baseline. However, the JWINS++ algorithm uses lossy compression for the floating point data. Therefore, it used about a third of the communication budget that the accumulation baseline used. Figure 16 shows all algorithms the amount of data that was exchanged with a neighbor. As seen in Section 5.5.7, the lossy compression impacts the achievable accuracy, which might explain why JWINS++ is surpassed by the accumulation baseline. Additionally, the parameter-wise averaging scheme has a larger impact on small , as was shown in Section 5.2. Hence, JWINS++ has less of an edge at 30% sharing.

Celeba. For the results of this section, the CNN model for the Celeba dataset was trained for 350 epochs. The resulting convergence plots are shown in Figure 17 for 10% sharing and in Figure 18 for 30% sharing. The highest accuracies of the algorithms with both budgets are listed in Table 12. For 10% sharing, all the partial model algorithms' testing loss begin to diverge. Since their training loss is also substantially lower than that of the 100% sharing baseline, the cause of the divergence is likely overfitting to the local dataset. Both JWINS++ and the TOPK accumulation algorithm begin to diverge around epoch 50, and subsampling begins to diverge a bit later. Shortly afterward, their testing accuracy also begins to stall out. In other words, with all these algorithms, the model can be fully trained in about 100 epochs compared to the 350 epochs

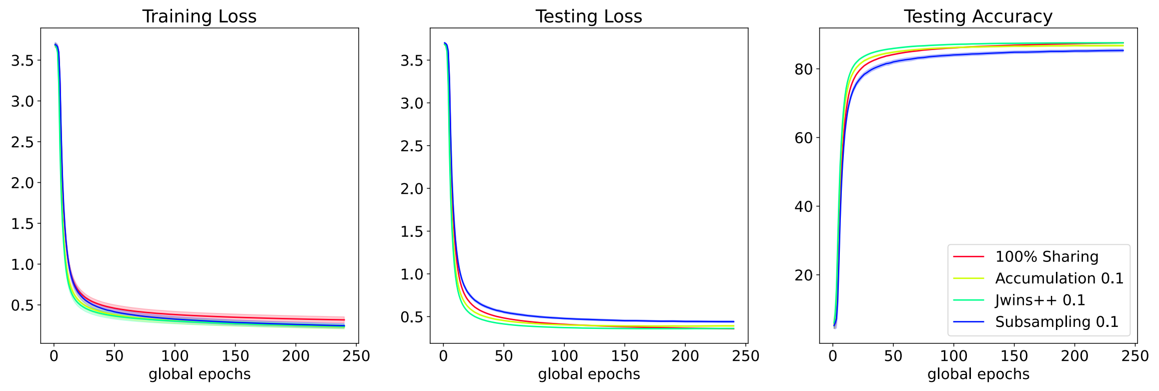

Figure 14: Training loss, testing loss, and testing accuracy for 240 epochs of the Femnist CNN model. JWINS++ with an α of 0.1 is compared against Subsampling, Accumulation and 100% Sharing.

Algorithm	Avg	Std
Accumulation 0.3	87.703	0.103
Jwins++ 0.3	87.679	0.099
Jwins++ 0.1	87.572	0.151
100% Sharing	87.553	0.076
Accumulation 0.1	86.774	0.288
Subsampling 0.3	86.664	0.252
Subsampling 0.1	85.341	0.424

Table 11: Highest average accuracies and their standard deviations averaged over 3 runs for the Femnist CNN model.

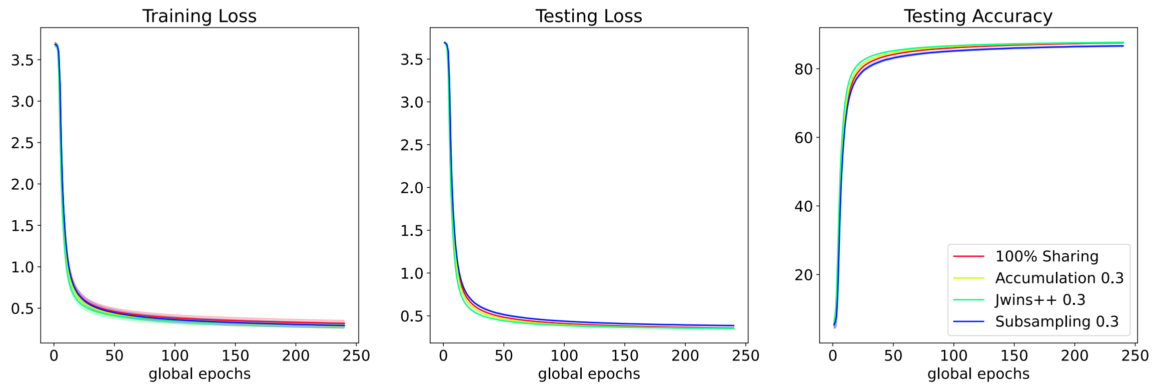

Figure 15: Training loss, testing loss, and testing accuracy for 240 epochs of the Femnist CNN model. JWINS++ with an α of 0.3 is compared against Subsampling, Accumulation and 100% Sharing.

needed for full sharing. However, their final accuracies are 0.45 points for JWINS++, 0.94 points for TOPK accumulation, and 1.65 for subsampling lower than full sharing. Unsurprisingly, for 30% sharing, all the partial model algorithms begin to diverge later, and as it already happened for the Femnist dataset, the TOPK accumulation algorithm beats JWINS++. Hence, JWINS++ benefits again much less from the increased communication budget.

Reddit. The Reddit dataset is evaluated over 50 epochs during which it fully converged. The convergence plots are shown in Figure 19 for 10% sharing and in Figure 20 for 30% sharing. The highest accuracies of the algorithms are listed in Table 13. For 10% sharing, both JWINS++ and the TOPK accumulation algorithm begin diverging on the test loss after about 15 epochs. As already seen in the plots for celeba, the divergence for JWINS++ is much faster. One possible

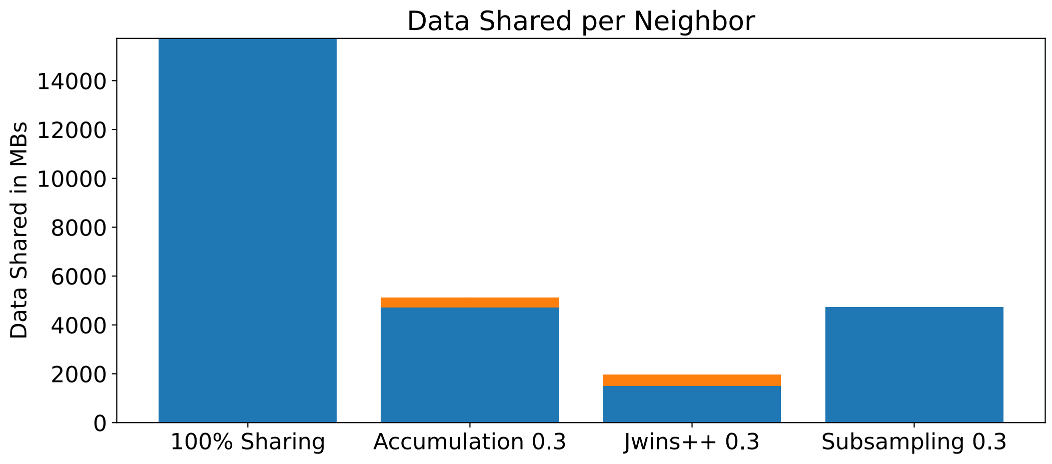

Figure 16: Data exchanged with each neighbor for the 100% sharing, Accumulation, JWINS++, and Subsampling algorithms. The latter two algorithms shared 30% of their parameters in each communication round.

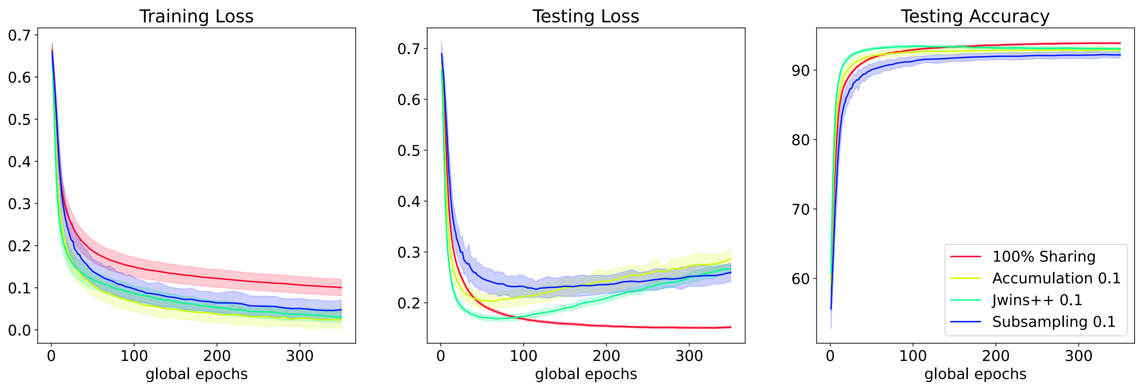

Figure 17: Training loss, testing loss, and testing accuracy for 350 epochs of the Celeba CNN model. JWINS++ with an α of 0.1 is compared against Subsampling, Accumulation and 100% Sharing.

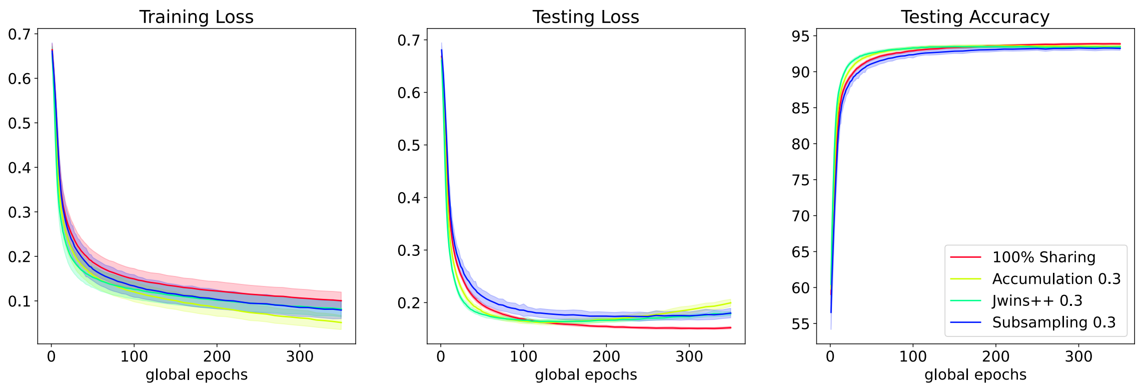

Figure 18: Training loss, testing loss, and testing accuracy for 350 epochs of the Celeba CNN model. JWINS++ with an α of 0.3 is compared against Subsampling, Accumulation and 100% Sharing.

explanation could be that the divergence is accelerated due to the faster convergence speeds of the parameter-wise averaging scheme. Nonetheless, JWINS++ beats the other two partial sharing methods in terms of the highest achieved accuracy. However, these arethe worst results so far for JWINS++ with a 0.69 accuracy percentage point deficit compared to 100% sharing. Also, Reddit is the first dataset with a substantial cap between subsampling and the others. It begins

Algorithm	Avg	Std
100% Sharing	93.889	0.130
Accumulation 0.3	93.608	0.189
Jwins++ 0.3	93.502	0.177
Jwins++ 0.1	93.436	0.211
Subsampling 0.3	93.295	0.289
Accumulation 0.1	92.948	0.379
Subsampling 0.1	92.244	0.422

Table 12: Highest average accuracies and their standard deviations averaged over 3 runs for the Celeba CNN model.

to diverge almost immediately on the testing loss. Finally, Reddit is also the first dataset in which the training losses are substantially lower for all partial sharing models suggesting substantial overfitting. For 30% sharing, all the partial model sharing algorithms begin to diverge later. As it happened for Femnist and Celeba, JWINS++ is again beaten by the TOPK accumulation baseline in terms of its highest accuracy. However, compared to the other two datasets, benefited more from the higher communication budget with an increase in accuracy of 0.55 percentage points.

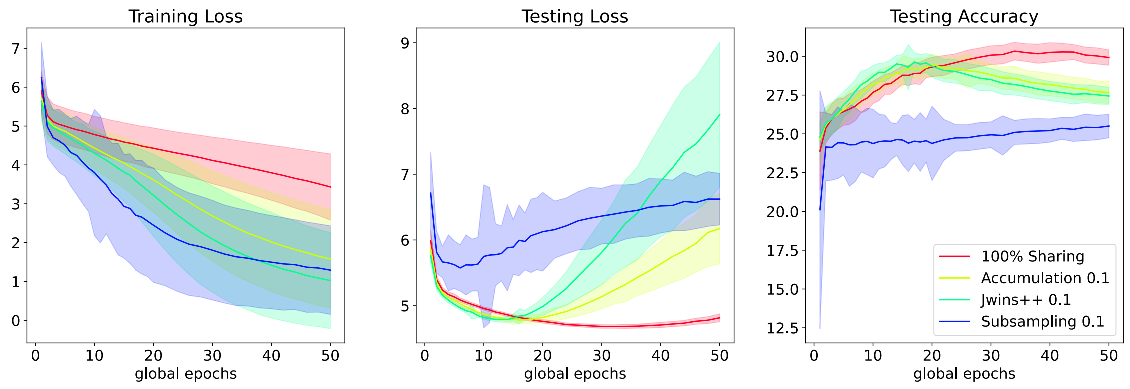

Figure 19: Training loss, testing loss, and testing accuracy for 50 epochs of the Reddit RNN model. JWINS++ with an α of 0.1 is compared against Subsampling, Accumulation and 100% Sharing.

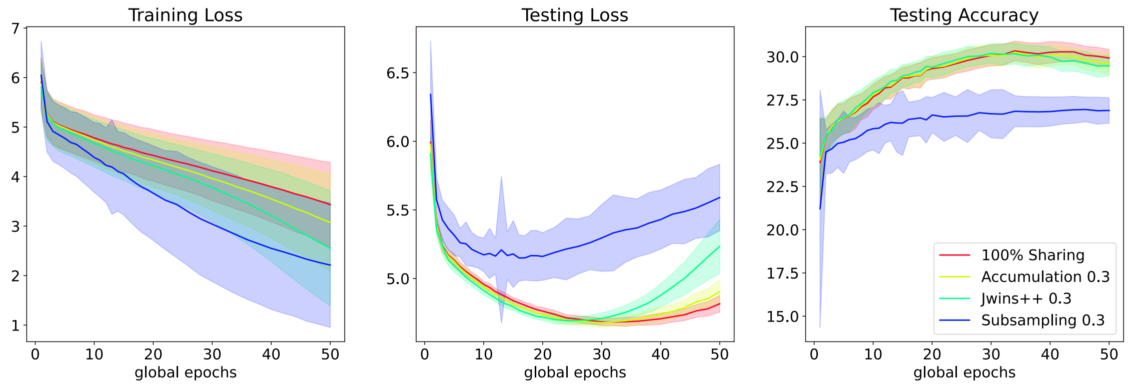

Figure 20: Training loss, testing loss, and testing accuracy for 50 epochs of the Reddit RNN model. JWINS++ with an α of 0.3 is compared against Subsampling, Accumulation and 100% Sharing.

Shakespeare. The Shakespeare dataset was trained for 120 epochs and reached full convergence. The convergence plots are shown in Figure 21 for 10% sharing and in Figure 22 for 30%

Algorithm	Avg	Std
100% Sharing	30.333	0.574
Accumulation 0.3	30.257	0.569
Jwins++ 0.3	30.188	0.571
Jwins++ 0.1	29.639	0.602
Accumulation 0.1	29.471	0.687
Subsampling 0.3	26.947	0.713
Subsampling 0.1	25.505	0.760

Table 13: Highest average accuracies and their standard deviations averaged over 3 runs for the Reddit RNN model.

sharing and Table 14 list the highest accuracies.
For 10% sharing, the two partial sharing baselines' test loss begins to diverge almost immediately. However, the testing accuracy begins to improve again after an initial drop. The JWINS++ algorithm begins to diverge after around 20 epochs and does not come close to the lowest testing loss of full sharing. These stark differences between the partial sharing models for the testing loss are contrasted by the training loss curves, which follow the same trajectory. The training loss curves show that the partial sharing models begin to overfit from the beginning compared to 100% sharing. A possible explanation could be that the model changes significantly initially, and partial sharing cannot propagate the changes fast enough. Lastly, the testing accuracy for JWINS++ beats the other partial model sharing algorithms by a wide margin of 3.45 compared to the TOPK accumulation algorithm. However, it is also 2.24 accuracy percentage points behind full sharing.
For 30% sharing, there are improvements everywhere. There is a substantially smaller gap between the partial and full sharing algorithms for the training loss. However, they still begin to diverge between 30 and 40 global epochs. For the highest testing accuracy, the familiar picture presents itself again. The TOPK accumulation algorithm beats JWINS++, but JWINS++ still convergences slightly faster at the beginning of the training.

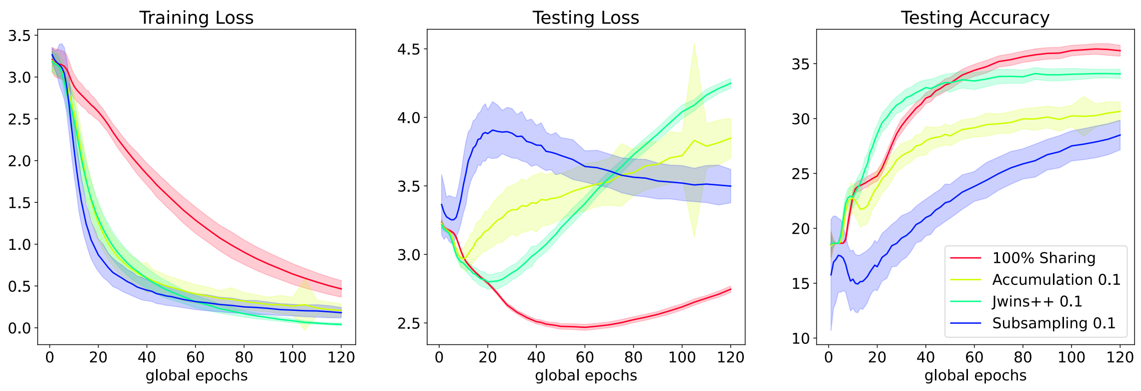

Figure 21: Training loss, testing loss, and testing accuracy for 120 epochs of the Shakespeare RNN model. JWINS++ with an α of 0.1 is compared against Subsampling, Accumulation and 100% Sharing.

Cifar10. The Cifar10 dataset was trained for 500 epochs and converged fully for the 100% sharing algorithm. The dataset has been the toughest to train as its data distribution is the most non-iid. Every node stored four randomly drawn shards, each containing at most data belonging to two classes. The results for 10% sharing are shown in Figure 23, the results for 30% in Figure 24. A list with the highest accuracies per algorithm is presented in Table 15.
The results for Cifar10 differ from all previous ones. The first notable difference is that at

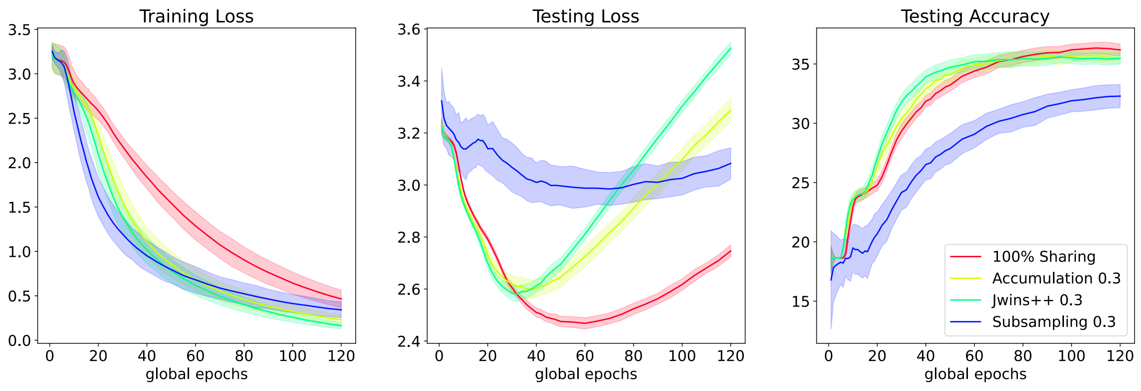

Figure 22: Training loss, testing loss, and testing accuracy for 120 epochs of the Shakespeare RNN model. JWINS++ with an α of 0.3 is compared against Subsampling, Accumulation and 100% Sharing.

Algorithm	Avg	Std
100% Sharing	36.342	0.483
Accumulation 0.3	35.715	0.544
Jwins++ 0.3	35.594	0.488
Jwins++ 0.1	34.099	0.547
Subsampling 0.3	32.286	0.978
Accumulation 0.1	30.654	0.847
Subsampling 0.1	28.498	1.357

Table 14: Highest average accuracies and their standard deviations averaged over 3 runs for the Shakespeare RNN model.

both 10% and 30% sharing, the TOPK accumulation baseline is performing worse than the subsampling baseline in terms of testing loss and accuracy. For 10% sharing, the accumulation algorithm's test loss begins to stall after only 30 epochs, while the loss improves further for subsampling. However, both algorithms follow the same trajectory for their test accuracy, which is slowly improving. They are contrasted by JWINS++, which starts like full sharing from a substantially lower test loss and manages to keep up with full sharing for the first 30 epochs. Subsequently, it begins to diverge as well. These oddities are also reflected in the test accuracy, the gap between JWINS++ and the other partial sharing baselines is more than ten percentage points, and between it and full sharing, it is 3.88 points. Those are the highest difference among all the experiments. The picture is not much different for 30% sharing. The partial model algorithms begin to diverge on their testing loss a bit later and reach higher accuracies. The cap between JWINS++ and subsampling also narrowed a bit, as the former benefited less from the increased communication budget. Another curiosity is that there is not much difference in the training losses between these algorithms compared to, for example, Reddit or Shakespeare. However, the results for the testing set are substantially different.

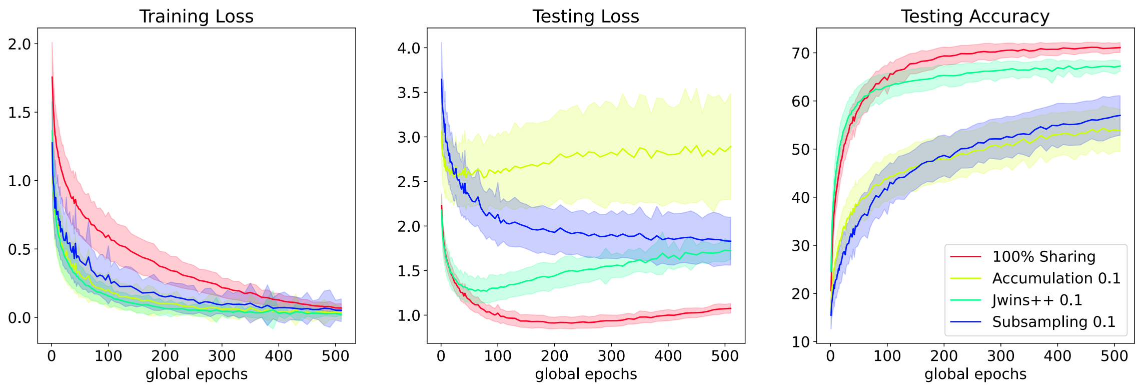

Figure 23: Training loss, testing loss, and testing accuracy measured druing the training of LeNet on the Cifar10 dataset. JWINS++ with an α of 0.1 is compared against Subsampling, Accumulation and 100% Sharing.

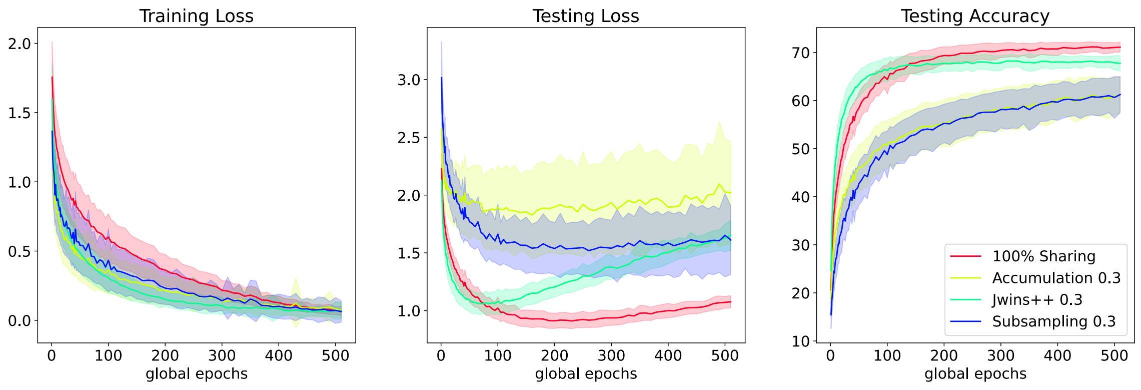

Figure 24: Training loss, testing loss, and testing accuracy measured druing the training of LeNet on the Cifar10 dataset. JWINS++ with an α of 0.3 is compared against Subsampling, Accumulation and 100% Sharing.

Algorithm	Avg	Std
100% Sharing	71.152	1.0395
Jwins++ 0.3	68.226	1.121
Jwins++ 0.1	67.268	1.085
Subsampling 0.3	61.233	3.722
Accumulation 0.3	61.139	3.887
Subsampling 0.1	56.988	4.143
Accumulation 0.1	54.341	4.713

Table 15: Highest average accuracies and their standard deviations averaged over 3 runs for the LeNet model trained on Cifar10.

6 Dynamically Changing Topologies

In the previous section, methods to reduce the communication overhead were examined. However, all of them worked by reducing the amount of shared data. The goal of this section is twofold. Firstly, it tests whether different communication patterns, such as random walks or dynamically changing typologies, can achieve the same accuracies in fewer communication rounds. Secondly, it aims to show that JWINS++ works in dynamic environments.

6.1 Asynchronous Random Walks

Dynamic topologies are widely used in decentralized learning. For example, a node can use a peer sampling service [27] to decide to which node the current weights are sent. Randomized Gossip algorithms [7] have also been used for consensus averaging, where a node randomly selects a neighbor with which it shares data. These algorithms also support completely asynchronous implementations. For example, random walks have also been used in asynchronous settings with token gossip learning [27], where a node responds to a received message by averaging it with its weight and subsequently forwarding the new weight reactively to a new neighbor. However, the number of reactive messages is limited by a token account. Additionally, a node sends periodic proactive messages to a random neighbor.

In order to evaluate JWINS++ in gossip learning settings, an asynchronous version of the D-PSGD algorithm is used. Similar to the token gossip learning algorithm, proactive messages are sent periodically. For our algorithm, this happens directly after the periodic averaging step that processes all received models. Reactive messages are sent without first averaging them with the local model and only if they have not yet run out of fuel. The fuel is a replacement for the token accounts in token gossip learning and makes the messages fixed size random walks. It limits the size of the random walk to its initial fuel. The received weights are stored in a list and averaged with the local weight once the local SGD steps are finished. The pseudocode of this algorithm is given in Algorithm 12. The procedure COMMLOOP is called asynchronously. It handles the receiving of the models and the reactive forwarding of them. The messages are tagged with a fuel count and a history of all visited nodes. The latter is used to sample a new neighbor with SAMPLENEIGHBOR to avoid sending it to a node that already received the message. The procedure MODELAVERAGING averages all the received weights. It is called in the main loop after the local SGD step. At the end of the main loop, a proactive random walk message is sent to a randomly chosen neighbor.

Algorithm 13 shows the Random Walk algorithm adapted to JWINS++. One significant departure from JWINS++ as described in Algorithm 11 is that accumulation is handled differently. As the proactive messages are sent at the end of the main loop, the accumulation can account for changes due to SGD and averaging in one step. The differences to the Random Walk Algorithm 12 are that the only partial models in the wavelet domain are shared.

The actual implementation used for the experiments differs slightly from Algorithm 12. It waits after the local SGD step for a dynamically changed period to regulate the incoming message's lag, i.e., if the node is ahead of others, the interval will be lengthened, and if it is behind, it will be shortened. Additionally, the proactive messages at the end of the main loop are only sent only with probability p.

6.2 Dynamic Graph

In real networks, nodes can connect and disconnect from each other. However, all experiments in Section 5 have been conducted on fixed communication topologies. On the other hand, changing communication topologies have been used by many decentralized learning systems [46, 48, 2]. This section presents a simple extension of D-PSGD to dynamically changing topologies. The algorithm corresponds closely to the description of D-PSGD given in Algorithm 4. It differs only in that at the end of the main loop, every C iterations a random walk message is sent that advertises the node to the recipient of the random walk. The recipient will subsequently establish a connection. It is based on random walk-based sampling methods **??**. Consequently, the weight matrix W needs to be updated as well. Each node has a lower and upper bound for the number of neighbors. It will disconnect from a random neighbor when the number of

Algorithm 12 Random Walk (RW)

```
Require: The algorithm is run on all $N$ nodes in parallel. Every node $i \in [N]$ initializes the
    weight $x_i^{(0,0)} = x_0$. The local dataset $D_i$, the learning rate $\eta$, the number of communication
    steps $T$, the number of local SGD steps $S$, the set of neighbors $E_i$, lower bound fraction $\alpha_{low}$,
    and the start fuel $F_0$
 1: procedure CommLoop($Q_i$)
 2:     while true do
 3:         $(x_j, F, H) \leftarrow receivefromNeighbor$
 4:         $Q.append((x_j))$
 5:         if $F > 0$ then
 6:             $k \leftarrow$ SampleNeighbor($E_i, H$)
 7:             send $(x_j, F - 1, H \cup \{i\})$ to neighbor $k$
 8: procedure ModelAveraging($x_i$, $Q_i$)
 9:     $a_i \leftarrow \mathbf{0}$
10:     for $x_j \in Q_i$ do
11:         $a_i \leftarrow a_i + x_j$
12:     $a_i \leftarrow a_i + x_i$
13:     return $a_i$
14: $Q_i \leftarrow \emptyset$
15: CommLoop($Q_i$)                                                    ▷ Asynchronous call
16: for $t \leftarrow 0$ to $T$ do
17:     $x_i^{(t+\frac{1}{2},S)} \leftarrow$ LocalSGD($x_i^{(t,0)}$)
18:     $x_i^{(t+1,S)} \leftarrow$ ModelAveraging($x_i^{(t+\frac{1}{2},S)}, Q$)
19:     $k \leftarrow$ SampleNeighbor($M_i, \{i\}$)
20:     send $(Z_i^{(t+1,S)}[I_i], F_0\{i\})$ to neighbor $k$
```

Algorithm 13 JWINS++ Random Walk

Require: The algorithm is run on all N nodes in parallel. Every node $i \in [N]$ initializes the weight $x_i^{(0,0)} = x_0$. The local dataset D_i, the learning rate η, the number of communication steps T, the number of local SGD steps S, the set of neighbors E_i, the fraction of parameters α to share, the accumulation vector $V_i^0 = \mathbf{0}$, counting vector $c_i = \mathbf{0}$, lower bound fraction α_{low}, set of neighbors M_i, and the start fuel F_0

```
 1: procedure CommLoop($Q_i$)
 2:     while true do
 3:         $(x_j[I_j], F, H) \leftarrow receivefromNeighbor$
 4:         $Q.append((x_j[I_j]))$
 5:         if $F > 0$ then
 6:             $k \leftarrow$ SampleNeighbor($M_i, H$)
 7:             send $(x_j[I_j], F - 1, H \cup \{i\})$ to neighbor $k$
 8: procedure PartialModelAveraging($x_i$, $Q_i$)
 9:     $A_i \leftarrow \mathbf{0}$
10:     $C_i \leftarrow \mathbf{0}$
11:     for $(x_j[I_j]) \in Q_i$ do
12:         $a \leftarrow x_i$
13:         $a[I_j] \leftarrow x_j[I_j]$
14:         $C[I_j] \leftarrow C[I_j] + 1$
15:         $A_i \leftarrow A_i + a$
16:     $Q_i \leftarrow \emptyset$
17:     $A_i \leftarrow A_i + x_i$
18:     $A_i \leftarrow A_i / C_i$
19:     return $A_i$
20: $Q_i \leftarrow \emptyset$
21: $Z_i^{(0,S)} \leftarrow$ DWT($x_i^{(0,0)}$)
22: CommLoop($Q_i$)                                        ▷ Asynchronous call
23: for $t \leftarrow 0$ to $T$ do
24:     $x_i^{(t+\frac{1}{2},S)} \leftarrow$ LocalSGD($x_i^{(t,0)}$)
25:     $Z_i^{(t+\frac{1}{2},S)} \leftarrow$ DWT($x_i^{(t+\frac{1}{2},S)}$)
26:     $Z_i^{(t+1,S)} \leftarrow$ PartialModelAveraging($Z_i^{(t+\frac{1}{2},S)}, Q$)
27:     $V_i^{t+1} \leftarrow V_i^t + (Z_i^{(t+1,S)} - Z_i^{(t,S)})$
28:     $I_i \leftarrow$ BoundedTopK($|V_i^{t+1}|, \alpha \cdot \text{len}(x_i^{(t,0)})$)
29:     $V_i^{t+1}[I_i] \leftarrow \mathbf{0}$
30:     $x_i^{(t+1,0)} \leftarrow \text{DWT}^{-1}(Z_i^{(t+1,0)})$
31:     $k \leftarrow$ SampleNeighbor($M_i, \{i\}$)
32:     send $(Z_i^{(t+1,S)}[I_i], F_0\{i\})$ to neighbor $k$
```

neighbors surpasses the upper bound and refuses a disconnect request when it would fall below the lower bound. Exact implementation details are given in the Section 3.3.1 in the chapter about *TCPRandomWalkRouting*. The algorithm is implemented in the *SharingDynamicGraph* module.

The JWINS++ algorithm for the dynamic communication topology implements the same changes discussed in the previous paragraph on top of the Algorithm 11. Its implementation is given in the JwinsDynamicGraph module.

6.3 Evaluation

The two algorithms introduced in this section are evaluated on the Reddit dataset. The random walk and dynamic graph algorithms are evaluated with full model sharing and partial model sharing with JWINS++. The hyperparameters used to train the RNN model on the Reddit dataset are the same as in the previous section and can be found in Tabel 1. The methodology used for training and evaluating the algorithms is described in Section 4. As described there, all shown results are averages over three runs initiated with a new seed.

6.3.1 Asynchronous Random Walks

The Reddit dataset is evaluated over 160 epochs during which it fully converged. Therefore, it is three times slower than in the synchronous setting. Figure 25 depicts the convergence plots for the Random Walk algorithm. It shows the results of the RW algorithm for different probabilities p at which the random walk is sent. The length of the random walk was limited to four. Hence for $p = 1$, the RW algorithm used the same communication budget per round as D-PSGD used for Reddit in the experiments in Section 5.5.8.

Of the four experiments depicted in Figure 25 the experiment named *No Communication* corresponds to training the model only on local data, i.e., no communication with neighbors is initiated. Its test loss begins to diverge immediately, and its accuracy does not improve either. However, it has the lowest training loss, a clear sign of overfitting to the local dataset. The other three experiments have a probability of 0.25, 0.5, and 1 to initiate a proactive random walk. In the beginning, their testing loss and accuracy follow the same trajectory. However, at around epoch 50, they begin to separate. The two experiments with lower probabilities for proactive RW messages begin to fall behind, and their testing loss and accuracy begin diverging. For both of them, their training loss is lower than the one for $p = 1$, suggesting they started overfitting too much to the local dataset. Table 16 shows the highest accuracies achieved by these methods. Unsurprisingly, the more RW walks were initiated, the higher the final accuracy was. The table also lists the results for training the CNN model of Reddit for 50 iterations with D-PSGD. Its results are only surpassed for the RW with $p = 1$ but at the cost of three times iterations. Another notable difference between these two learning paradigms is that the standard deviation is much larger for the RW setting. Hence the individual results obtained at every node are less stable for RW. Nonetheless, these results show that incorporating information from distant nodes into the training process can improve final accuracy.

Algorithm	Avg	Std
RW p=1	30.73	1.269
100% Sharing	30.333	0.574
RW p=0.5	29.557	1.538
RW p=0.25	28.378	1.360
No Communication	24.187	1.707

Table 16: Highest average accuracies and their standard deviations averaged over 3 runs for the Reddit CNN model.

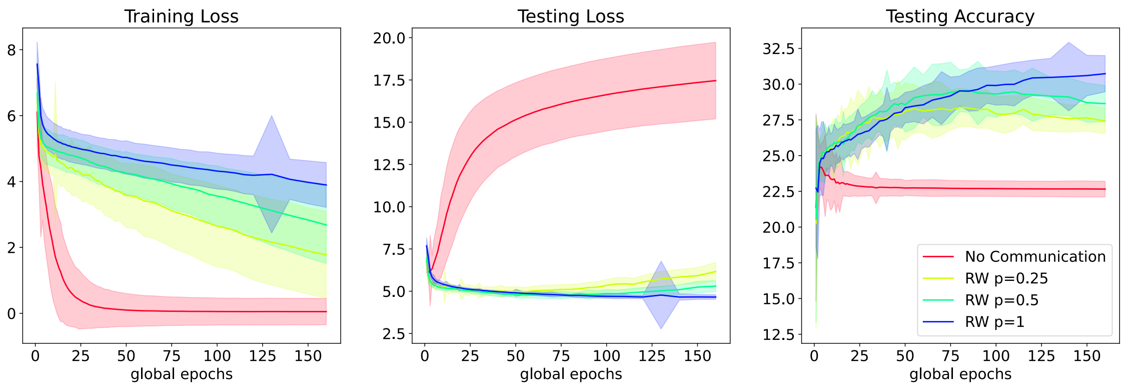

Figure 25: Training loss, testing loss, and testing accuracy for 160 epochs of the Reddit trained with the RW algorithm. The results for four different probabilities with which RW messages are sent proactively are shown.

The results of experiments for the JWINS++ version of the RW algorithm are depicted in Figure 26. The JWINS++ algorithm for RW (Algorithm 13) is compared against the RW algorithm (Algorithm 12). All experiments were conducted with $p = 1$, i.e., after every local training round, a RW message is sent to a random neighbor. The JWINS++ algorithm was evaluated for sharing 10% and 30% of the model. At the beginning of training, the JWINS++ algorithm for 10% sharing pulls ahead of the others for testing loss and accuracy. However, it begins to diverge after about 75 epochs and achieves a much lower final accuracy, as seen in Table 17. The results for JWINS++ with 30% sharing closely follow the convergence trajectory of the RW baseline. At the end of the training, its maximal accuracy was only 0.13 percentage points lower. Unlike the 10% sharing algorithm, its training loss is also not substantially lower than the RW baseline.

Comparing the results for JWINS++ in the RW setting and the synchronous setting (See Table 13) shows that JWINS++ at 10% sharing and 30% sharing achieved similar results compared to their respective baselines. In other words, the gap between the final accuracies of JWINS++ and full sharing are similar in both settings. Therefore, JWINS++ is effective in setting where weights from new nodes are received regularly without the need to keep a cached version of neighbors' weights like other communication-efficient DL algorithms [58, 33, 60].

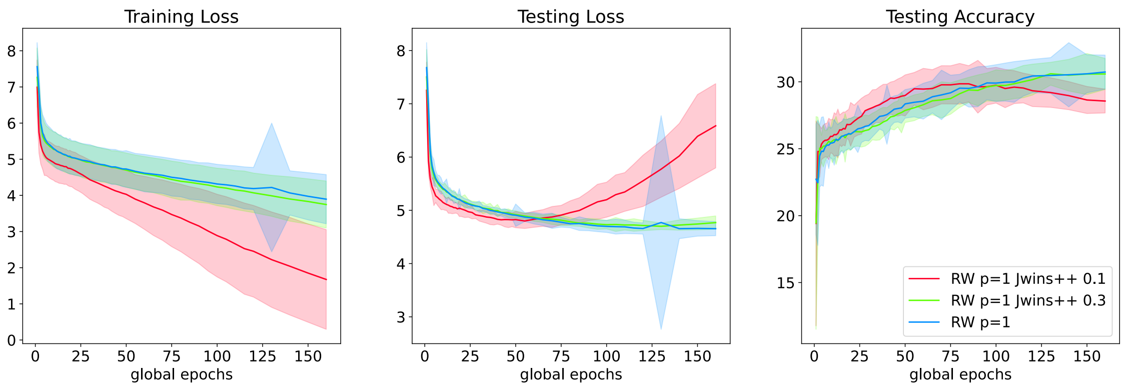

Figure 26: Training loss, testing loss, and testing accuracy for 50 epochs of the Reddit RNN model. The results for with α values of 0.1 and 0.3 are compared against the RW algorithm without compression.

Algorithm	Avg	Std
RW p=1	30.73	1.269
RW p=1 Jwins++ 0.3	30.603	0.854
RW p=1 Jwins++ 0.1	29.853	1.519

Table 17: Highest average accuracies and their standard deviations averaged over 3 runs for the Reddit CNN model.

6.3.2 Dynamic Graph

CNN for Reddit has been evaluated over 50 epochs on the dynamic topology and fully converges by the end. The topology changes at the end of every epoch. On average, every node establishes one new connection and closes another one. A node can have a minimum of 3 and a maximum of 5 neighbors. The convergence plots for the Reddit RNN model are depicted in Figure 27. It shows the results of the JWINS++ for 10% and 30% sharing, as well as , i.e., full sharing, on the dynamic graph and compares them against the full sharing results on the static graphs from the experiments in Section 5.5.8. The results on the dynamic graphs are very similar to the ones on static graphs (See Figures 19 and 20). JWINS++ for 10% sharing diverges in both cases after 15 epochs in the testing loss and a bit later on the accuracy. The training loss follows a similar trajectory as well. For 30% sharing, the divergence happens a bit later on the dynamic topology. For 100% sharing, the trajectories are almost the same. However, full sharing on the dynamic topology is consistently slightly ahead of its static counterpart. The best accuracies are presented in Table 18. Full sharing on the dynamic topology beats its pendent on the static topology by 0.48 percentage point, and even JWINS++ for 30% sharing surpasses it by 0.43 points. On the dynamic topology, the cap between JWINS++ at 30% sharing and fully sharing is only 0.05 points, while for 10% sharing, it is 0.92. These correspond roughly to the gaps measured on the static topology. Hence, we conclude that JWINS++, with its heuristics for parameter sharing, also works on dynamic typologies and that there is no need for cached weights to do decentralized learning communication efficiently.

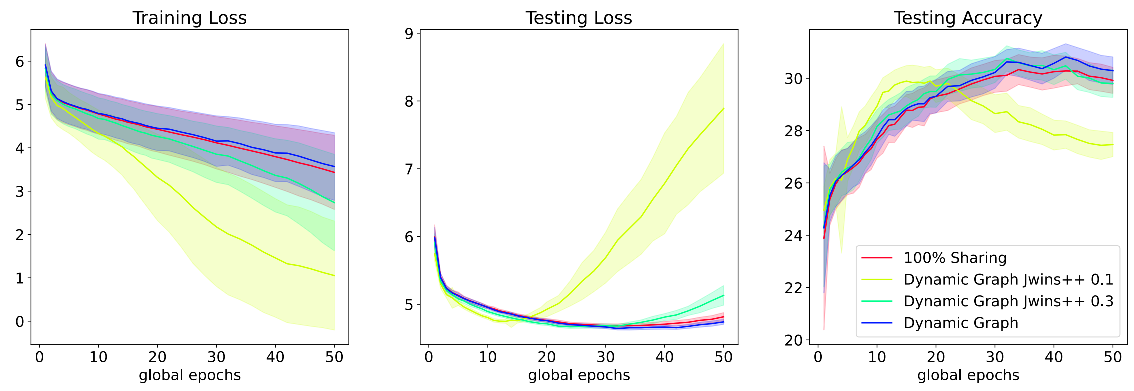

Figure 27: Training loss, testing loss, and testing accuracy for 50 epochs of the Reddit trained on a dynamic Topology. The results for 100% sharing as well as with α values of 0.1 and 0.3 on the dynamic topology are compared against 100% sharing on the static topology.

Algorithm	Avg	Std
Dynamic Graph	30.812	0.509
Dynamic Graph Jwins++ 0.3	30.761	0.482
100% Sharing	30.333	0.574
Dynamic Graph Jwins++ 0.1	29.895	0.567

Table 18: Highest average accuracies and their standard deviations averaged over 3 runs for the Reddit CNN model.

7 Conclusion

This thesis presented a new algorithm for communication efficient decentralized machine learning called Jwins++. It consists of four parts core parts: Accumulation of model changes to build an importance score, representation of the model in the wavelet frequency domain, parameter-wise averaging, and lossy floating point compression. These four parts were evaluated on the Femnist dataset and proved effective.

A thorough evaluation of this hybrid algorithm on five different datasets and models showed to be a well-performing algorithm. Moreover, it proved incredibly effective when only 10% of the model was shared. In this scenario, Jwins++ dominated over the subsampling and accumulation baselines, despite sharing even less data than them due to lossy model compression. Together, these compression methods reduced the model size by 20x compared to full sharing. Furthermore, for three out of the five datasets, its final accuracy came within one percentage point of the full sharing baseline. Only for Cifar10 did it not come within one percentage point when 30% of the model parameters were shared. However, Cifar10 is also the dataset for which it beat the other two partial model sharing baselines by the widest margin.

Jwins++ also proved effective in decentralized machine learning with random walks or dynamically changing communication topologies. Hence, Jwins++ is well suited for real-world decentralized learning applications. Its success in such environments also sets it apart from communication-efficient DL algorithms that rely on one replicas [58, 33, 60]. These algorithms would need to share a full copy of their replica with every new neighbor, while Jwins++ shares the same partial model it sends to all other neighbors.

Our experiments have shown that Jwins++ is a viable solution for communication efficient decentralized learning. As all runs of Jwins++ were not tuned separately, there should be room to improve its performance with better tuned hyperparameters and custom learning rates schedules, which could stop the early divergence observed in so many experiments.

References

[1] Dan Alistarh et al. "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/6c340f25839e6acdc73414517203f5f0-Paper.pdf.

[2] Mahmoud Assran et al. "Stochastic Gradient Push for Distributed Deep Learning". In: *CoRR* abs/1811.10792 (2018). arXiv: 1811.10792. URL: http://arxiv.org/abs/1811.10792.

[3] Tal Ben-Nun and Torsten Hoefler. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis*. 2018. DOI: 10.48550/ARXIV.1802.09941. URL: https://arxiv.org/abs/1802.09941.

[4] Luca Boccassi et al. *ZeroMQ: An open-source universal messaging library*. 2022. URL: https://zeromq.org.

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. "Optimization Methods for Large-Scale Machine Learning". In: *SIAM Review* 60.2 (2018), pp. 223–311. DOI: 10.1137/16M1080173. eprint: https://doi.org/10.1137/16M1080173. URL: https://doi.org/10.1137/16M1080173.

[6] Léon Bottou et al. "Stochastic gradient learning in neural networks". In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 12.

[7] S. Boyd et al. "Randomized gossip algorithms". In: *IEEE Transactions on Information Theory* 52.6 (2006), pp. 2508–2530. DOI: 10.1109/TIT.2006.874516.

[8] Pádraig Brady et al. *Crudini 0-9.3*. 2019. URL: https://github.com/pixelb/crudini.

[9] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[10] Haitao Guo C. Sidney Burrus Ramesh Gopinath. *Wavelets and Wavelet Transforms*. OpenStax CNX, Aug. 2015, pp. 5–36. URL: http://cnx.org/contents/110bab92-1948-4958-b1c7-8fc0926c392c@5.16.

[11] Haitao Guo C. Sidney Burrus Ramesh Gopinath. *Wavelets and Wavelet Transforms*. OpenStax CNX, Aug. 2015, pp. 37–46, 199–206. URL: http://cnx.org/contents/110bab92-1948-4958-b1c7-8fc0926c392c@5.16.

[12] Sebastian Caldas et al. *LEAF: A Benchmark for Federated Settings*. 2019. arXiv: 1812.01097 [cs.LG].

[13] Thomas Cassimon et al. "Designing resource-constrained neural networks using neural architecture search targeting embedded devices". In: *Internet of Things* 12 (2020), p. 100234. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2020.100234. URL: https://www.sciencedirect.com/science/article/pii/S2542660520300676.

[14] Yann Collet. *LZ4*. 2022. URL: http://www.lz4.org/.

[15] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[16] *DecentralizePy*. SaCS. Nov. 9, 2021. URL: https://gitlab.epfl.ch/sacs/decentralizepy (visited on 06/17/2022).

[17] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[18] Tim Dettmers. *8-Bit Approximations for Parallelism in Deep Learning*. 2015. DOI: 10.48550/ARXIV.1511.04561. URL: https://arxiv.org/abs/1511.04561.

[19] Nikoli Dryden et al. “Communication Quantization for Data-Parallel Training of Deep Neural Networks”. In: *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. 2016, pp. 1–8. DOI: 10.1109/MLHPC.2016.004.

[20] P. Elias. “Universal codeword sets and representations of the integers”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pp. 194–203. DOI: 10.1109/TIT.1975.1055349.

[21] P. Erdös and A. Rényi. “On Random Graphs I”. In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[22] Jonas Geiping et al. “Inverting Gradients - How easy is it to break privacy in federated learning?” In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 16937–16947. URL: https://proceedings.neurips.cc/paper/2020/file/c4ede56bbd98819ae6112b20ac6bf145-Paper.pdf.

[23] *General Data Protection Regulation*. European Union. Apr. 27, 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 05/30/2022).

[24] Suyog Gupta et al. “Deep Learning with Limited Numerical Precision”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France: JMLR.org, 2015, 1737–1746.

[25] F. Maxwell Harper and Joseph A. Konstan. “The MovieLens Datasets: History and Context”. In: *ACM Trans. Interact. Intell. Syst.* 5.4 (2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[26] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[27] István Hegedűs, Gábor Danner, and Márk Jelasity. “Decentralized learning works: An empirical comparison of gossip learning and federated learning”. In: *Journal of Parallel and Distributed Computing* 148 (2021), pp. 109–124. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2020.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0743731520303890.

[28] Franck Iutzeler et al. “Asynchronous distributed optimization using a randomized alternating direction method of multipliers”. In: *52nd IEEE Conference on Decision and Control*. 2013, pp. 3671–3676. DOI: 10.1109/CDC.2013.6760448.

[29] A. Jadbabaie, Jie Lin, and A.S. Morse. “Coordination of groups of mobile autonomous agents using nearest neighbor rules”. In: *IEEE Transactions on Automatic Control* 48.6 (2003), pp. 988–1001. DOI: 10.1109/TAC.2003.812781.

[30] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[31] Sian Jin et al. “DeepSZ”. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2019. DOI: 10.1145/3307681.3326608. URL: https://doi.org/10.1145%2F3307681.3326608.

[32] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[33] Anastasia Koloskova, Sebastian Stich, and Martin Jaggi. “Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3478–3487. URL: https://proceedings.mlr.press/v97/koloskova19a.html.

[34] Jakub Konečný et al. “Federated Learning: Strategies for Improving Communication Efficiency”. In: *NIPS Workshop on Private Multi-Party Machine Learning*. 2016. URL: https://arxiv.org/abs/1610.05492.

[35] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “The CIFAR-10 dataset”. In: 55.5 (2014). URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[36] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[37] Gregory Lee et al. *PyWavelets/pywt: PyWavelets 1.3.0.* Version v1.3.0. Mar. 2022. DOI: 10.5281/zenodo.6347505. URL: https://doi.org/10.5281/zenodo.6347505.

[38] Mu Li. “Scaling Distributed Machine Learning with the Parameter Server”. In: *Proceedings of the 2014 International Conference on Big Data Science and Computing.* BigDataScience ’14. Beijing, China: Association for Computing Machinery, 2014. ISBN: 9781450328913. DOI: 10.1145/2640087.2644155. URL: https://doi.org/10.1145/2640087.2644155.

[39] Xiangru Lian et al. “Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems.* NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, 5336–5346. ISBN: 9781510860964.

[40] Yujun Lin et al. “Deep gradient compression: Reducing the communication bandwidth for distributed training”. In: *arXiv preprint arXiv:1712.01887* (2017).

[41] Peter Lindstrom. “Fixed-Rate Compressed Floating-Point Arrays”. In: *IEEE Transactions on Visualization and Computer Graphics* 20 (Aug. 2014). DOI: 10.1109/TVCG.2014.2346458.

[42] Peter Lindstrom and Martin Isenburg. “Fast and Efficient Compression of Floating-Point Data”. In: *IEEE transactions on visualization and computer graphics* 12 (Sept. 2006), pp. 1245–50. DOI: 10.1109/TVCG.2006.143.

[43] Ryan McDonald, Keith Hall, and Gideon Mann. “Distributed Training Strategies for the Structured Perceptron”. In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics.* HLT ’10. Los Angeles, California: Association for Computational Linguistics, 2010, 456–464. ISBN: 1932432655.

[44] Brendan McMahan and Daniel Ramage. *Federated Learning: Collaborative Machine Learning without Centralized Training Data.* Google. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[45] Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics.* Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1273–1282. URL: https://proceedings.mlr.press/v54/mcmahan17a.html.

[46] Zeyu Meng et al. “Learning-Driven Decentralized Machine Learning in Resource-Constrained Wireless Edge Computing”. In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications.* 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488817.

[47] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *ICML.* 2010, pp. 807–814. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[48] Angelia Nedić and Alex Olshevsky. “Distributed Optimization Over Time-Varying Directed Graphs”. In: *IEEE Transactions on Automatic Control* 60.3 (2015), pp. 601–615. DOI: 10.1109/TAC.2014.2364096.

[49] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[50] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[51] Frank Seide et al. “1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs”. In: *Interspeech 2014*. 2014. URL: `https://www.microsoft.com/en-us/research/publication/1-bit-stochastic-gradient-descent-and-application-to-data-parallel-distributed-training-of-speech-dnns/`.

[52] C.E. Shannon. “Communication in the Presence of Noise”. In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. DOI: `10.1109/JRPROC.1949.232969`.

[53] Rishi Sharma et al. “Get More for Less in Decentralized Learning Systems”. In: *Under review* (2022).

[54] Reza Shokri and Vitaly Shmatikov. “Privacy-preserving deep learning”. In: *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 2015, pp. 909–910. DOI: `10.1109/ALLERTON.2015.7447103`.

[55] A. Skodras, C. Christopoulos, and T. Ebrahimi. “The JPEG 2000 still image compression standard”. In: *IEEE Signal Processing Magazine* 18.5 (2001), pp. 36–58. DOI: `10.1109/79.952804`.

[56] Alexander Smola and Shravan Narayanamurthy. “An Architecture for Parallel Topic Models”. In: *Proc. VLDB Endow.* 3.1–2 (2010), 703–710. ISSN: 2150-8097. DOI: `10.14778/1920841.1920931`. URL: `https://doi.org/10.14778/1920841.1920931`.

[57] Nikko Ström. “Scalable Distributed DNN Training Using Commodity GPU Cloud Computing”. In: *Interspeech 2015*. 2015. URL: `https://www.amazon.science/publications/scalable-distributed-dnn-training-using-commodity-gpu-cloud-computing`.

[58] Hanlin Tang et al. “Communication Compression for Decentralized Training”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: `https://proceedings.neurips.cc/paper/2018/file/44feb0096faa8326192570788b38c1d1-Paper.pdf`.

[59] Martin Vetterli and Cormac Herley. “Wavelets and Filter Banks: Theory and Design”. In: *Signal Processing, IEEE Transactions on* 40 (Oct. 1992), pp. 2207 –2232. DOI: `10.1109/78.157221`.

[60] Haozhao Wang et al. “Error-Compensated Sparsification for Communication-Efficient Decentralized Training in Edge Environment”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (2022), pp. 14–25. DOI: `10.1109/TPDS.2021.3084104`.

[61] Linnan Wang et al. “FFT-Based Gradient Sparsification for the Distributed Training of Deep Neural Networks”. In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’20. Stockholm, Sweden: Association for Computing Machinery, 2020, 113–124. ISBN: 9781450370523. DOI: `10.1145/3369583.3392681`. URL: `https://doi.org/10.1145/3369583.3392681`.

[62] Jeffrey Wigger. *FastVarInts*. 2022. URL: `https://github.com/jw-96/FastVarInts`.

[63] Lin Xiao and S. Boyd. “Fast linear iterations for distributed averaging”. In: *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*. Vol. 5. 2003, 4997–5002 Vol.5. DOI: `10.1109/CDC.2003.1272421`.

[64] Lin Xiao, Stephen Boyd, and Seung-Jean Kim. “Distributed average consensus with least-mean-square deviation”. In: *Journal of Parallel and Distributed Computing* 67.1 (2007), pp. 33–46. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2006.08.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731506001808`.