**Scalable Computing Systems Laboratory (SaCS):**

# Model sharing and communication techniques in decentralized systems

**School of Computer and Communication Sciences, EPFL Lausanne, Switzerland**

*№321682:* Nevena Drešević

*Project Advisor:* Professor Anne-Marie Kermarrec
*Project Supervisors:* Dr. Rafael Pires, Mr. Rishi Sharma

Optional Semester Project

June 10, 2022

**Abstract**

Decentralized machine learning systems are essential in providing scalability while preserving the privacy and security of the data. They do so without needing a central coordinator and by leveraging distributed computing. These systems are applicable in healthcare, finance, social media platforms, and other areas, primarily focusing on industries holding sensitive data. Therefore, their performance is of huge importance, and improvements need to be made. This report aims to present a decentralized system's behavior when put through different communication protocols, machine learning model parameter sharing techniques, and its two model modification methods. We examine the performance of implementing communication in which nodes exchange information with all neighbors (D-PSGD) versus randomized communication (RWM). Additionally, we analyze those algorithms with various approaches for picking model parameters to share, as well as how to update the local model. The system is tested on three different datasets: a recommendation one (MovieLens Latest) and two image classification ones (CelebA, FEMNIST). The obtained results mostly show the domination of the D-PSGD algorithm, even with limited processing time, but with a significant trade-off regarding network load. Hence, the report results also provide a place for further improvements and research.

# Contents

# Chapter 1

# Introduction

Today, there are various challenges to traditional centralized learning practices. As an immense amount of data is generated constantly and rapidly at different locations, centralized systems have significant scalability limitations. Also, the data being located only in one place, being collected from different resources, or being moved to other sites increases doubts about its privacy and security. However, decentralized learning techniques can elevate these problems. A decentralized approach is much more natural in recommender systems or systems where data is linked to a user. Here, the security and privacy concerns are solved by allowing only the owner of sensitive data to access it and perform further modifications. Furthermore, the data is distributed in multiple locations, allowing for faster processing, a more scalable system, and better handling of highly distributed or imbalanced data.

In decentralized machine learning, the distributed nodes have models trained only on local datasets. Besides local training, the models are improved by exchanging model parameters, such as weights and biases, with their neighbors and updating their own model. In that way, nodes are gradually improving their model and, therefore, the models of their neighbors. This machine learning approach enables the creation of a robust and scalable model without sharing data, thus addressing the critical issue of data privacy. However, sharing model parameters increases network traffic since models are usually large, and combining neighboring models with local ones increases the time and requires more computations. Hence, new questions about which model parameters should be shared, what should be the fraction of the model, with whom, and how to do the local model updates require additional research.

This report addresses previously mentioned issues with decentralized machine learning systems and researches the system's performances using different communication and model sharing techniques. We vary the communication of a node with all its neighbors or with a randomly chosen one, alternating how to update the local model. Furthermore, the selection of model parameters varies, such as choosing the ones that changed the most. The goal is to analyze the system's performance in terms of model accuracy, network traffic loads, and total processing time. In order to examine, two different machine learning models are used on three different datasets, covering both recommendation and classification models.

Firstly, in Chapter 2, we explain two machine learning models used, one for recommendation and the other for classification, and give a brief overview of decentralized learning. Then, in Chapter 3, we present how all components are connected and together make the decentralized system. Additionally, we explain two different techniques for exchanging information in the network and the corresponding local model updates. Here, the details of various sharing methods are also depicted, as well as an explanation for configuring the system. Next, Chapter 4 covers datasets, together with metrics used for system evaluation and test environment specifics. Furthermore, various experiments are presented to understand the performance of different communication and model sharing techniques. This report is ended by providing a conclusion and future work in Chapter 5.

# Chapter 2

# Background

This chapter describes the two machine learning models used in the system: a matrix factorization model that falls into the category of recommendation models and a convolutional neural network that we use for category of classification models. Furthermore, decentralized learning is explained.

## 2.1 Matrix Factorization

The first machine learning model, matrix factorization, is used in a recommendation system to predict the "rating" or "preference" a user would give to an item. It can be seen in various use cases, from suggesting buyers items that could interest them to suggesting users text, image, and video content that matches their preferences. It belongs to a class of collaborative methods that extract similarities and make future predictions based on past interactions between users and items (user-item interaction matrix).

The matrix factorization method only relies on user-item interaction information and assumes a latent model to explain these interactions. It decomposes the user-item interaction matrix $A \in \mathbb{R}^{n \times m}$ into a product of two matrices of lower dimension $X \in \mathbb{R}^{n \times k}$ and $Y \in \mathbb{R}^{m \times k}$ representing embeddings that summarise user tastes and items profiles respectively. The embeddings are learned such that the product $XY^T$ is a good approximation of the feedback matrix $A$. The $(i, j)$ entry of $XY^T$ is simply the dot product $< X_i, Y_j >$ of the embeddings of $user_i$ and $item_j$. To make correct predictions, this entry needs to be close to $A_{i,j}$. Difficulties arise as matrix $A$ is often very sparse, with only some user-item interactions being known. An additional regularization parameter $\lambda$ can be added to improve the model performance and avoid overfitting. Moreover, much of the variation in interaction data are due to effects associated with either users or items, known as biases, independent of any interactions. The intuition behind this is that some users give lower or higher ratings than others, and some items receive lower or higher ratings than others systematically. Hence, the bias vectors $b \in \mathbb{R}^{n \times 1}$ and $c \in \mathbb{R}^{m \times 1}$ are included. Formally, including all terms, the loss function which the model minimizes can be written as follows:

$$J(X, Y, b, c) =$$
$$\frac{1}{2} \sum_{(i,j) \in I} (a_{ij} - b_i - c_j - \sum_{l=1}^{k} x_{il} y_{jl})^2 + \frac{\lambda}{2} ||X||^2 + \frac{\lambda}{2} ||Y||^2$$

where $I$ represents the set of indices for known values in $A$. Upon learning matrix $X$ and matrix $Y$, the predictions for $user_i$ and $item_j$ are obtained as:

$$p_{ij} = X_i \cdot Y_j + b_i + c_j$$

## 2.2 CNN (Convolutional Neural Network)

Convolutional Neural Networks (CNN) are deep learning algorithms that can take in an input image, assign the importance of various aspects and objects, and differentiate them based on the image's learnable weights and biases. Compared to other classification algorithms, CNN requires much less pre-processing. Besides images, they can be used for video, speech, audio signal inputs, etc. Biological processes inspired convolutional networks in that the pattern of connections between neurons resembles that observed in the animal visual cortex. The brain's cortex responds to visual stimuli only in an area called the receptive field, where individual neurons reside. The receptive fields of different neurons partially overlap such that they cover the entire visual area.

The architecture of CNN consists of 3 main types of layers. The first type of layer is a convolutional layer, a core layer, which requires input data, filer, and feature map. It applies a convolution operation to the input, passing the result to the next layer by converting all the pixels in its receptive field into a single value. Then, a pooling layer represents downsampling and performs dimensionality reduction. Similarly to a convolutional layer, it applies a filter across the entire input but without weights and reduces the number of input parameters. The final layer is the fully connected (FC) layer, where each node in the output layer connects directly to a node in the previous layer. The FC layer performs classification with features extracted through the previous layers and their filters. An example of CNN model architecture that performs the classification of a handwritten digit is shown in Figure 2.1. The CNN becomes more complex with each layer, whereas earlier layers focus on simple features such as colors and edges. As the image data progresses through the CNN layers, the model begins to recognize more prominent elements of the image until it finally identifies the object.

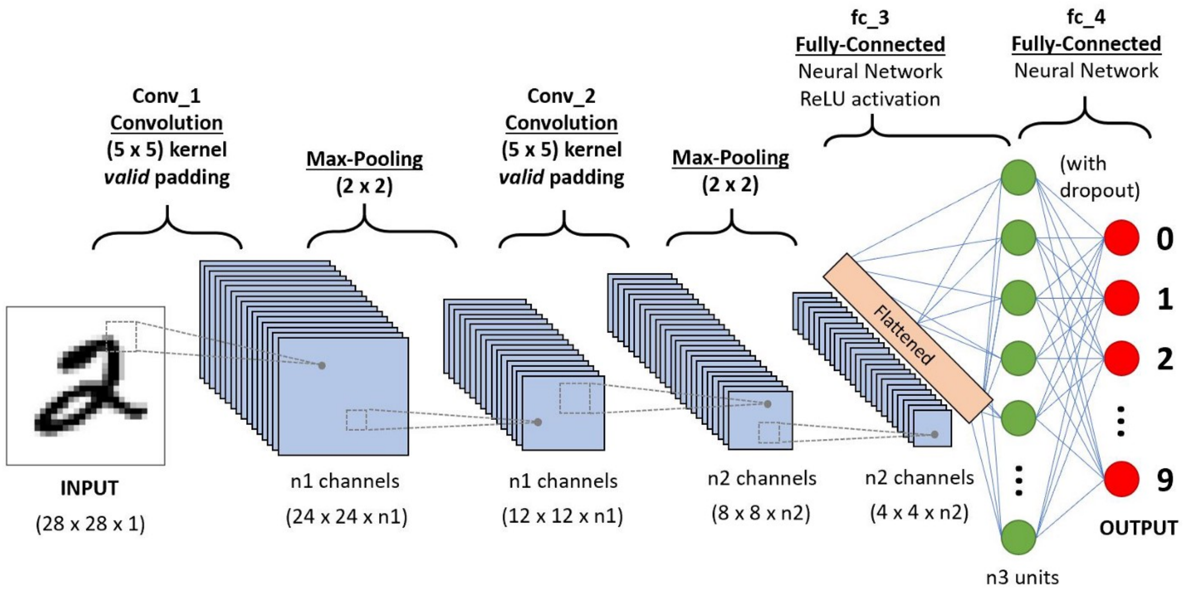

Figure 2.1: CNN model sequence to classify handwritten digits. [1]

## 2.3 Decentralized Learning

In a complex ecosystem of servers where data are inherently distributed and can be sensitive or high-volume, decentralized machine learning techniques are most helpful. They enable collaborative model training without sharing raw data and can be used to combine all local learnings from inherently distributed datasets into a single machine learning model. These algorithms are called gossip algorithms, in which each device only communicates with its neighbors without requiring an aggregation server or any central component.

For instance, in a recommender system, the number of items and users grows by the minute; hence one of the main challenges of centralized recommenders remains their scalability. Therefore, decentralized systems can be proposed for recommendation purposes. Typically, we assume that nodes are connected according to a specific network topology in such a system. After performing the local learning task, each node forwards some information to its neighbors. For example, this information can be the output of the local learning tasks in decentralized learning systems or users' profiles. Relying on such a gossiping protocol enables the data or model to be disseminated in the network until convergence is reached. Training data is fully distributed to the nodes in the system, i.e., raw data is produced by users who hold a single personal profile record. Such approaches let the users' data stay where it is produced, thus limiting their exposure. Nevertheless, models need to be aggregated in order to provide relevant recommendations for unseen items. Learning tasks are performed on local data on user devices and merged in the decentralized learning system through a gossip-based protocol. Model aggregation and exchanging information between neighbors can have various approaches and implementations based on the system's needs.

# Chapter 3

# Implementation

This section first describes the architecture of the system, followed by different communication methods between the nodes in the network and model averaging. Next, the four methods for sharing model parameters are presented. Lastly, in Section 3.4, we present the system configuration for running specific datasets, the machine learning model, its optimization and parameters tuning, as well as which communication and sharing technique are to be used.

## 3.1 System Architecture

The system consists of multiple components that encapsulated produce a configurable decentralized system. It supports different network sizes and node interconnections, the dataset and machine learning model used, various model sharing, and neighboring communication schemes.

At the start of the run, the nodes are initialized, and each holds the data of several users. Each node's initially distributed local data points are disjunct, and nodes perform training only on their local data set. The received dataset is split into a train set, used to improve the model, and a test set to check whether the model outputs accurate values compared to the ground truth. These two datasets are created by taking a corresponding ratio from each user's data samples. Then, each node can start epochs of training locally and share information with other nodes depending on the model sharing method and communication technique specified. This training step typically consists of passing data through the machine learning model, calculating the gradient, back-propagation, and the optimizer step. After each training step on local data, a node waits or sends model information. Depending on the communication technique used, a node updates its local model with model parameters received from other neighbors. Finally, the system continuously provides evaluation metrics of each local model, which later enables us to process the final performance as an average of all nodes.

## 3.2 Communication

For communication inside the nodes network, we can use one of the two algorithms that differ in the number of neighbors they exchange model with and correspondingly model updates. The first algorithm is D-PSGD, where each node communicates and exchanges information with all its neighbors in each training epoch. A node receives model parameters from all its neighbors, and weights attributed to each model are calculated using the Metropolis-Hastings protocol. On the other hand, the RWM algorithm randomly chooses a neighbor it sends its model to and continues local training only upon receiving a message from a neighbor.

### 3.2.1 D-PSGD

Decentralized SGD (Stochastic Gradient Descent) is the application of solving the optimization problem of finding minimum loss function by updating parameters one by one for each data point in a decentralized setting. In the Decentralized Parallel Stochastic Gradient Descent (D-PSGD) algorithm, nodes average their local model with those of all its neighbors after a step of training on local data. Then, each node sends its models and an integer corresponding to its degree (number of neighbors) to all neighbors. Upon receiving a model, it is merged with its own through a weighted average. For that, we use Metropolis-Hastings weights, where each model parameter $p_k$ in a node $k$ is computed by attributing weights to the contributions of its $N$ neighbors and that of itself, $p_k^{t+1} = w_k p_k^t + \sum_{i=1}^{N} w_i p_i^t$, with $w_i = \frac{1}{1+max(d_k,d_i)}$ and $w_k = 1 - \sum_{i=1}^{N} w_i$, where $d_i$ is the degree of node $i$. If a node has not received given model parameter, we consider only those of its neighbors, $w_k = 0$, and the weight attributed to each neighbor, just like Metropolis-Hastings, is inversely proportional to their degree, i.e., $w_i = \frac{1}{1+d_i \sum_{j=1,i\neq j}^{N} \frac{1}{d_j}}$.

### 3.2.2 RWM (Random Walk Model)

The second algorithm resembles the idea of random walks, a random process that describes a path that consists of a succession of random steps. Here, the communication choices of nodes are independent and random. In the Random Walk Model algorithm, nodes average their local model upon receiving a model from one neighbor after a step of training on local data. Hence, the communication is asynchronous as a node progresses to the next round only after receiving a model. The model update is done by weighted averaging the received model with the local one. The weight of a model corresponds to confidence in the trained model, represented as the model's age. This age depends on the number of local training steps a node completes and the age of the nodes from which it receives the model. The age is increased at each local training step and additionally updated to the age of the received neighbor model if it is older, i.e., its age is bigger. After model averaging, a node sends its updated model and an integer corresponding to its age to one neighbor selected uniformly at random.

Due to the asynchronous advancement of nodes to further rounds, the system terminates once the first node completes the intended number of iterations. However, when calculating the performance metrics of an obtained run, we use the data from rounds contained in all nodes, i.e., we take the minimum number of iterations reached by all nodes.

## 3.3 Sharing Methods

The system provides multiple choices of algorithms used for sharing information about the local model with other nodes. It is important to note that only the model parameters are shared. We use a couple of approaches to experiment between sharing the whole model, or only a fraction of it, as well as picking the suitable parameters to share.

### 3.3.1 Full Model

The *Full Model* sharing approach represents the most simple idea of sharing model parameters; the whole model is shared. Here, when a node needs to share its information, it sends all of the model parameters. The sharing configuration for this is the following:

```
sharing_package = decentralizepy.sharing.Sharing
sharing_class = Sharing
```

### 3.3.2 Partial Model

The second sharing method resembles sending only a fraction of the whole model. Nodes have specified float number $\alpha$ representing a fraction of the model parameters to share. Furthermore, picking which ones to send can be further adjusted by defining what changes in the model should be accumulated. In every communication round $\alpha$ fraction of the parameters that changed the most since the last time they were shared is chosen. Using *accumulation* means that the accumulation of local training changes measures changes. Additionally, using *accumulate_averaging_changes* adds to the accumulation the changes made due to the model averaging step. These accumulation choices can be specified in the configuration depicted below by setting corresponding values to *True* or *False*. The model changes can not be accumulated only based on model averaging; hence if any accumulation is to be done, the local changes must be included. In other words, setting *accumulate_averaging_changes* to *True* and *accumulation* to *False* has no effect. Also, in configuration, $\alpha$ is set to a float number between 0 or 1, with one meaning the full model should be shared (equivalent to *Full Model* sharing).

```
sharing_package = decentralizepy.sharing.PartialModel
sharing_class = PartialModel
alpha = 0.3
accumulation = True
accumulate_averaging_changes = True
```

### 3.3.3 Random Alpha Model

This sharing method extends the previously presented *Partial Model* by allowing multiple possible fractions of a model to be shared. In each round of sharing, a random value representing the model fraction is chosen from a list of possible fractions. This list is called *alpha_list*, and it contains float numbers between 0 and 1, representing possible $\alpha$ used in the run. The configuration for this sharing method is presented below, with *metadata_cap* defining the threshold for sharing the entire model, i.e., if the randomly chosen $\alpha$ from the list is bigger than *metadata_cap*, then the full model is shared.

```
sharing_package = decentralizepy.sharing.RandomAlpha
sharing_class = RandomAlpha
alpha_list = [0.1, 0.2, 0.3, 0.4, 1.0]
metadata_cap = 0.5
accumulation = True
accumulate_averaging_changes = True
```

### 3.3.4 Sub-Sampling Model

Sub-sampling is a method that reduces data size by selecting a subset of the original data. Hence, the sub-sampling sharing is a simple version of partial sharing where the subset of model parameters to share is chosen randomly without inspecting any model parameter changes. This method uses defined $\alpha$ to create a random binary mask utilized to determine the $\alpha$ fraction of model parameters to be shared. The configuration to use this sharing method is the following:

```
sharing_package = decentralizepy.sharing.SubSampling
sharing_class = SubSampling
alpha = 0.3
```

## 3.4 System Configuration

The system can be applied to different networks, datasets, machine learning algorithms, communication techniques, and sharing methods. All these choices are defined in more detail as configuration parameters of the system. Each configuration setup yields one unique experiment, and the results metrics are stored for each node of the system on the machine executing that process. To execute a run of the system, the following parameters are specified:

- **DATASET:** Specifies the dataset used for training and evaluation, the location of train and test data, as well as the class implementing the machine learning model. An example configuration is as follows:

```
dataset_package = decentralizepy.datasets.MovieLens
dataset_class = MovieLens
model_class = MatrixFactorization
train_dir = /mnt/nfs/shared/leaf/data/movielens
test_dir = /mnt/nfs/shared/leaf/data/movielens
```

- **OPTIMIZER PARAMETERS:** Defines which machine learning optimizer is used and the learning rate of the model.

```
optimizer_package = torch.optim
optimizer_class = SGD
lr = 0.25
```

- **TRAIN PARAMETERS:** The configuration of the training module for each node. It specifies *rounds* representing the number of iterations per training step, a Boolean parameter *full_epochs* indicating whether training is done on full dataset (*True*) or mini-batches (*False*), and *batch_size* defining the number of samples in one batch of machine learning model step. Also, the loss function of the machine learning model is defined here.

```
training_package = decentralizepy.training.Training
training_class = Training
rounds = 9
full_epochs = False
batch_size = 8
shuffle = True
loss_package = torch.nn
loss_class = MSELoss
```

- **COMMUNICATION:** Defines the implementation of network communication API with IP addresses of used machines. The addresses are provided as a path to the JSON file containing the mapping of the machine id to its IP address.

```
comm_package = decentralizepy.communication.TCP
comm_class = TCP
addresses_filepath = /home/dresevic/decentralizepy/ip_4Machines.json
```

- **SHARING:** The configuration of the sharing method used, described in Section 3.3, with additional parameter *random_sharing* indicating whether a model is shared only with one random neighbor or all neighbors. In other words, setting *random_sharing = True* means that communication uses *Random Walk Model*, otherwise *D-PSGD*.

# Chapter 4

# Evaluation

This section manifests the evaluation of the decentralized system in various setups and the experimental methodology employed. It provides a set of experiments examining different aspects of model sharing and communication. In each experiment, we describe its setup, followed by the results and corresponding assessment.

## 4.1 Methodology

Here, the three used datasets are described, followed by metrics employed for experiments evaluation, and specification of the test environment for running the experiments.

### 4.1.1 Datasets

The experiments are run using three distinct datasets: MovieLens, Femnist, and Celeba. They are all significantly different in size, with MovieLens being the smallest and Femnist the biggest. Also, the MovieLens dataset is used for recommendation models, while the other two represent image type input for classification models.

**MovieLens Latest** The MovieLens Latest [2] dataset consists of collections of ratings and free-text tagging made by users on a movie recommendation service MovieLens. The users' ratings correspond to how much they appreciated a given movie. The goal of the recommendation model is to predict which rating a user would give for a particular movie. There are in total 100 836 ratings, where the rating scale is from 0.5 to 5.0, incrementally on every 0.5. There are 610 users giving ratings across 9724 movies, where each has rated at least 20 movies. Both users and movies are represented only by an id, without any user's personal information or additional information for the movie.

**FEMNIST** FEMNIST [3] is a federated extended MNIST dataset built by partitioning the data of handwritten digits and characters based on their writer. The dataset consists of 3 550 users and 805 263 total samples, with an average of 226.83 samples per user. The written digit/character is centered to fit an image size of 28x28 pixels. The dataset is utilized in image classification models to answer correctly which digit or character is written by the user.

**CelebA** Another image classification dataset is CelebA [3]. It is a large-scale face attributes dataset of celebrities, each having at least five images. The size of each image is 84x84 pixels. There are 200 288 samples in total with 9 343 different users, with an average of 21.44 samples per user. This dataset is used as input to the classifier that tells whether a person in the image is smiling or not.

### 4.1.2 Metrics

In order to measure the performance and determine the trade-offs of an experiment done on our decentralized system, multiple evaluation metrics are used. The most important ones are model train and test errors, test accuracy, network traffic, and total processing time. For the MovieLens dataset, the primary evaluation metric is test loss, where we report RMSE (Root Mean Square Error). While for FEMNIST and CelebA that is test accuracy, measuring the percentage of correct predictions as it is most suitable for classification models.

### 4.1.3 Test Environment

To run our simulated experiments, we used machines labostrex127-130 (in the IC cluster at EPFL). All the servers have the same specifications with processor Intel Xeon E5-2630 v3 at 2.40GHz and 128GB of memory running Ubuntu 20.04.2 LTS kernel 5.4.0-72.

## 4.2 Experiments

In the following experiments, we use varied datasets presented in Section 4.1.1, multiple sharing methods depicted in Section 3.3, and two communication techniques explained in Section 3.2. After experimenting with different setups, we present here the final experiments that best describe observed behavior. The experiments are done on a 4-regular graph with 96 nodes, using four machines, each maintaining 24 processes. In Table 4.1 we present parameter configurations used in experiments. We depict the machine learning models used, their tuned hyper-parameters, and a few additional parameters related to sharing in the system. Some parameters are obtained by running grid-search in a centralized setting and choosing the ones giving the best performance in terms of loss (or accuracy). For MovieLens(1), FEMNIST, and CelebA configurations, the parameter *full_epochs* is set to *False*, meaning that the rounds are done on mini-batches. In contrast, for MovieLens(2), the local rounds are completed by training on the whole dataset.

| | MovieLens (1) | MovieLens (2) | FEMNIST | CelebA |
|---|---|---|---|---|
| ML Model | MatrixFactorization | MatrixFactorization | CNN | CNN |
| Optimizer | SGD | Adam | SGD | SGD |
| Loss | MSELoss | MSELoss | CrossEntropyLoss | CrossEntropyLoss |
| Learning Rate | 0.25 | 0.01 | 0.01 | 0.001 |
| Rounds | 9 | 5 | 47 | 8 |
| Batch Size | 8 | 8 | 16 | 8 |

Table 4.1: Parameter configurations used in experiments.

Both FEMNIST and CelebA use a convolutional neural network with different numbers and orders of convolutional, pooling, and fully connected layers. The activation function for both is $ReLU$, while the number of classes for FEMNIST and CelebA is 62 and 2, respectively. Due to the complexity of CNN for the FEMNIST dataset, the total number of model parameters is 1.6 million, making this by far the biggest model of the three. For the CelebA dataset, this number is 28 000. With both of the datasets, the training data is initially distributed to the nodes such that data samples are disjunct, and training is done on a local dataset. In contrast, evaluation is done on global test data for all the nodes in the system. For the matrix factorization model used with MovieLens Latest dataset, the embedding size for both users and movies is 20 yielding a model with 413 360 parameters, with user and movie biases included. The train and test data is set by splitting each user's rating samples to a 70 : 30 ratio. Each node receives a subset of users and its corresponding test and train samples. Model training is done on a local test set, while evaluation is done only on local test samples, unlike global testing used in the previous two datasets. This evaluation resembles the actual recommender system that focuses on learning the traits of its user and making recommendations specific to them. If the testing is done globally, the performance of the decentralized system is poor for the MovieLens dataset.

### 4.2.1 Full Model

The first experiment represents the fundamental comparison of two communication techniques, D-PSGD and RWM. This comparison is made by sharing the entire model. Since the network graph is 4-regular, each node has exactly four neighbors, and correspondingly in D-PSGD algorithm, a node receives 4 neighboring models in each round, after which it conducts an update to its local model. When using RWM, a node receives one model from a neighbor.

Running the decentralized system with the MovieLens dataset is examined for both versions of local training. When a round of train step for a node assumes training on the whole dataset, sharing the model with only one neighbor and updating it by weighted averaging is worse than sharing with all neighbors and doing Metropolis-Hastings. The performance measured in terms of test loss is shown in Figure 4.1 with an RWM loss of 0.7630 and D-PSGD loss of 0.6684 at their final iterations. The loss calculated for the training phase is 0.0817 for RWM, and the model starts slowly overfitting. On the other hand, test loss for D-PSGD is 0.0985, with a continuation of learning. When the experiment is evaluated on a different type of local training, i.e., a round representing training only on mini-batches (with parameters defined in Table 4.1 MovieLens(1), there are still signs of overfitting for the RWM algorithm. The test losses for both models are depicted in Figure 4.2, and they are equal to 0.9754 and 0.8908 for RWM and D-PSGD, respectively, at their last iterations. Again, the performance of D-PSGD is better for completed iterations, but here the difference is almost negligible. In this case of local training, both communication versions produce a better model in the long run, as the learning is slow but constant. This behavior further shows that training more on the local dataset before communicating and updating the model yields better results in the short run but prevents the model from learning and achieving its full potential in the long run. In other words, the model gives too much confidence to its own learning, which is contrary to the case when a round of local training is done on mini-batches and sharing the model is faster. Then, a vast number of iterations are needed to get the desired performance. Hence, both RWM and D-PSGD algorithms update their model based on neighboring ones many times, making the performance difference between them small. Due to each node having 4 neighbors, the amount of average data shared by a node is four times larger in D-PSGD communication than in RWM. This can be confirmed in Figure 4.3 for MovieLens dataset.

Next, the experiment of full model sharing is run with CelebA dataset, and the test accuracy for RWM and D-PSGD algorithms is presented in Figure 4.4. The D-PSGD model gives better results, giving final accuracy of 89.42% after 250 iterations. The variation of local model performances on nodes is more extensive with RWM, as the model update is done based on a randomly chosen neighbor and with less information than in D-PSGD. For the RWM, the final accuracy after 200 iterations is 86.35%, while for D-PSGD, the corresponding value is 87.85%. The situation with calculated losses during training and testing is similar, with D-PSGD giving slightly better results. As the CNN model for CelebA is the smallest, the data overhead in each iteration is smaller than with matrix factorization used with MovieLens, though the pattern remains the same, i.e., the total data shared per node is around 4 times larger using D-PSGD than using RWM.

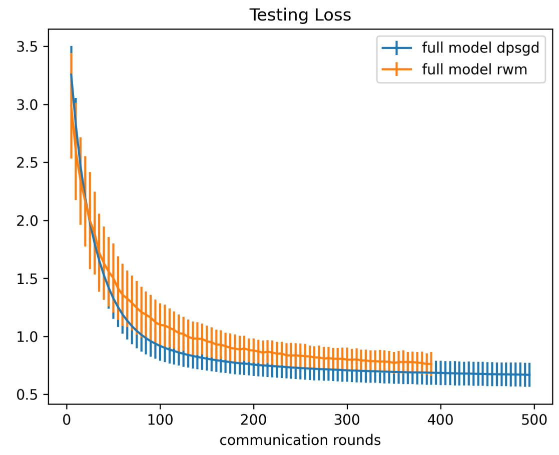

Figure 4.1: MovieLens: Test loss when training on the whole dataset.

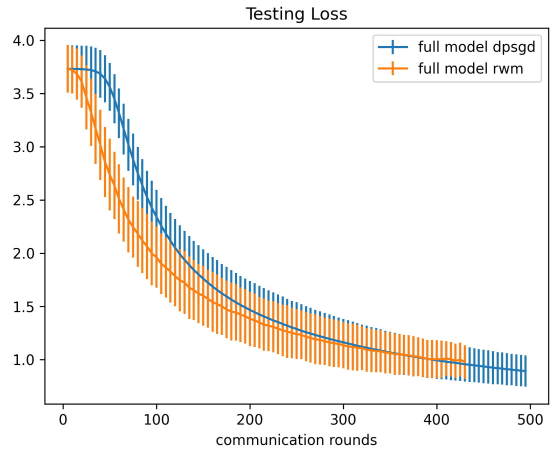

Figure 4.2: MovieLens: Test loss when training on mini-batches.

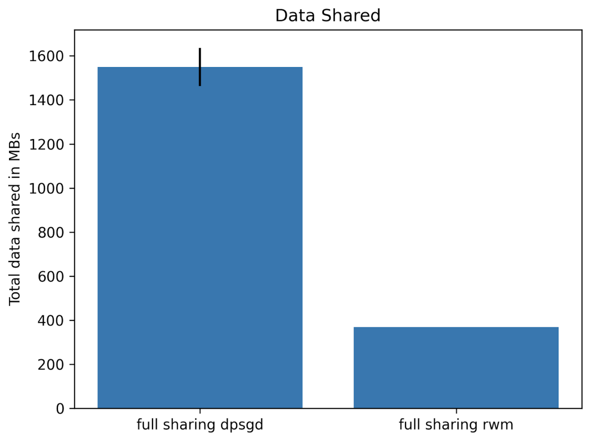

Figure 4.3: Average amount of data shared per node in D-PSGD and RWM (in MBs).

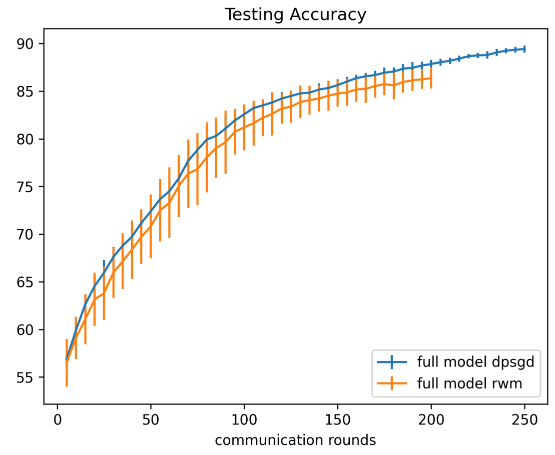

Figure 4.4: CelebA: Test accuracy when training on mini-batches and sharing full model.

### 4.2.2 Partial Model

Experiments done with different configurations for partial model sharing showed that the best performance is achieved when aggregation of model parameter changes is done both for local training changes and changes inducted by the model averaging step. Hence, for these experiments, parameters *accumulaiton* and *accumulate_averaging_changes* are set to *True*. In the following, observations done on MovieLens and CelebA are presented, with *MovieLens(1)* and *CelebA* configuration parameters defined in Table 4.1.

Testing the MovieLens dataset with partial model sharing for both RWM and D-PSGD is done with different values of $\alpha$, such as 0.1, 0.3, and 0.5. In both communication techniques, there is a clear pattern showing how the model performance and fraction of model shared are correlated; the less of a model is shared, the better the performance. This behavior is displayed in Figures 4.5 and 4.6 that show the model loss on evaluation, where we also present full sharing of the model, i.e., $\alpha = 1.0$, which verifies the previous statement. Furthermore, we compare the best models of both communication methods, those where 10% of the model parameters that changed the most are shared in each round, and plot the testing loss in Figure 4.7. It can be observed that RWM is learning more in the beginning until D-PSGD overtakes it around $140^{th}$ iteration. The D-PSGD continues to be better, and divergence becomes slightly more significant. The test loss for RWM in its final iteration (438) is 0.7460, while for D-PSGD, it is 0.6586 at that moment, leading to a final loss of 0.6243 in the $500^{th}$ iteration.

Next, analyzing the system's total processing time to complete the two runs leads us to the same conclusions. The total time taken for the run with RWM is 7.25 minutes, while D-PSGD takes 8.87 minutes. This difference is due to D-PSGD having more interactions between the nodes, plus each node averages its own model with multiple others, which also increases the computing time. If we observe the performance of D-PSGD given the same time as RWM used, then RWM gives worse results even though it completes more communication rounds. In Figure 4.8, we plot test loss comparison between D-PSGD and RWM given the same processing time, i.e., 7.25 minutes. For this time, the D-PSGD completes 384 iterations yielding a test loss of 0.6969, while that loss for RWM is 0.7460. Therefore, if processing time is taken into account, even though RWM is a faster algorithm, it is clear that D-PSGD produces a better model.

For our classification dataset, CelebA, the differences in model performance for various values of $\alpha$ are almost negligible, in contrast to bigger discrepancies observed for MovieLens. However, both for RWM and D-PSGD, the strongest learning and best accuracy are achieved when sharing 30% of model parameters. This sharing also achieves better performance compared to sharing the full model. The comparison of accuracy on the global testing dataset for both communication techniques is shown in Figure 4.9. Here, the distinction between RWM and D-PSGD is more prominent for model accuracy, as well as losses during training and testing. In $127^{th}$ iteration the RWM terminates with 84.01% model accuracy, while D-PSGD has 86.76% after completing 127 iterations and 88.87% in $170^{th}$ upon its termination. This pattern is also reflected in training loss, which takes values of 0.3506 and 0.2542 upon the termination of RWM and D-PSGD, respectively.

Again, in Figure 4.10, the time processing trade-off is depicted by showing test accuracy achieved when limiting execution time. Since D-PSGD consistently gives better results even when giving the same processing time to both algorithms when running this experiment, its shorter running time does not change the conclusion comparisons. The total time for RWM and D-PSGD to finish initial runs was 147.8 minutes and 194.2 minutes, respectively.

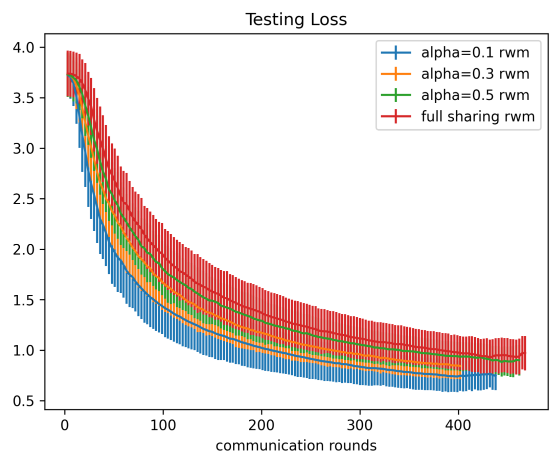

Figure 4.5: MovieLens: Test loss for partial sharing with RWM.

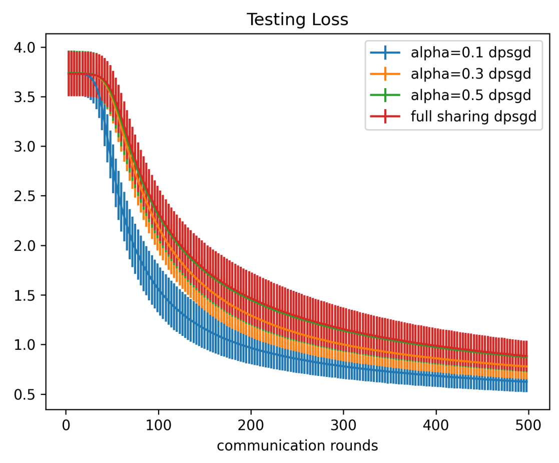

Figure 4.6: MovieLens: Test loss for partial sharing with D-PSGD.

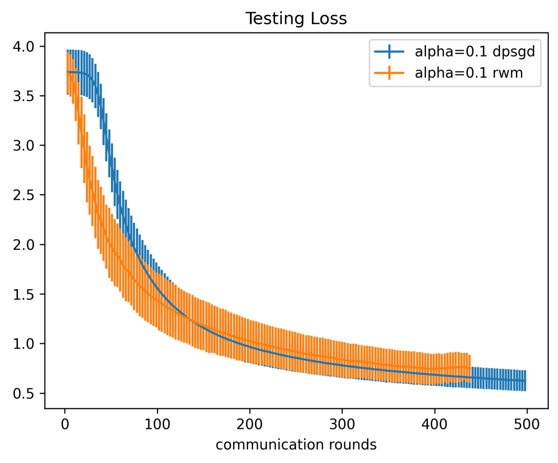

Figure 4.7: MovieLens: Test loss comparison between RWM and D-PSGD for $\alpha = 0.1$.

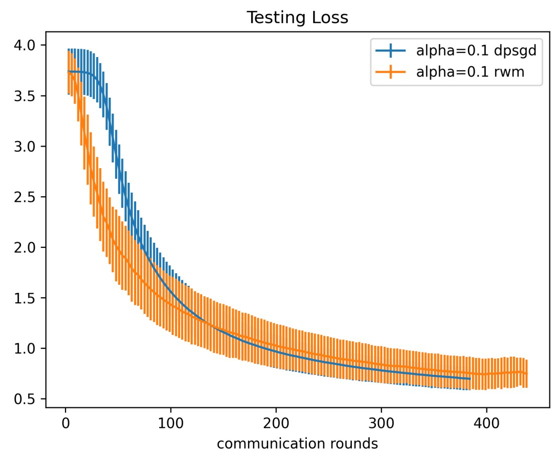

Figure 4.8: MovieLens: Test loss comparison given the same processing time.

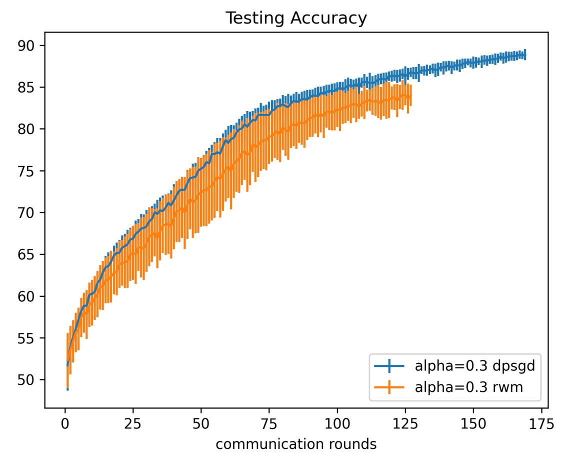

Figure 4.9: CelebA: Test accuracy comparison between RWM and D-PSGD for $\alpha = 0.3$.

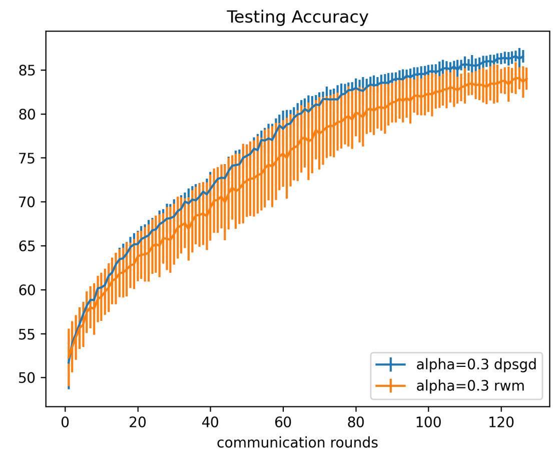

Figure 4.10: CelebA: Test accuracy comparison given the same processing time.

### 4.2.3 Random Alpha Model

This experiment further expands partial model sharing by randomly choosing a fraction of the model to share from $[0.1, 0.2, 0.3, 0.4, 1.0]$ in each round, with setting *metadata_cap* to 0.5. The choice of exact model parameters stays the same, meaning that the ones that changed the most, both locally and after averaging, are selected.

For the MovieLens dataset, the experiment was run with 5000 iterations with the testing loss during the time depicted in Figure 4.11. The differences between RWM and D-PSGD are minimal, again with already seen behavior. Only at the start, the random walk model is slightly better, while soon that changes leading to a final loss of 0.3503 in 4650th iteration. For D-PSGD, that value is 0.3234, while it finishes with 0.3189. The evaluation done on the training dataset shows that D-PSGD consistently gives slightly higher errors than RWM, from which it can be concluded that RWM focuses extensively on its local dataset and does not generalize as well as D-PSGD. The training loss values in iteration 4650 are 0.0215 for RWM and 0.0244 for D-PSGD, which then terminates with 0.0229.

In experiments done with CelebA dataset and CNN model, the D-PSGD model consistently performs better, with its advantage periodically changing over time. This divergence is shown with obtained accuracy on the testing set for both techniques presented in Figure 4.12. In iteration 195 RWM achieves 87.17% accuracy, D-PSGD 89.38% with increase to 90.84% in its 250th iteration. For errors on local testing, the situation remains consistent, D-PSGD has better results, and the models slowly diverge more and more.

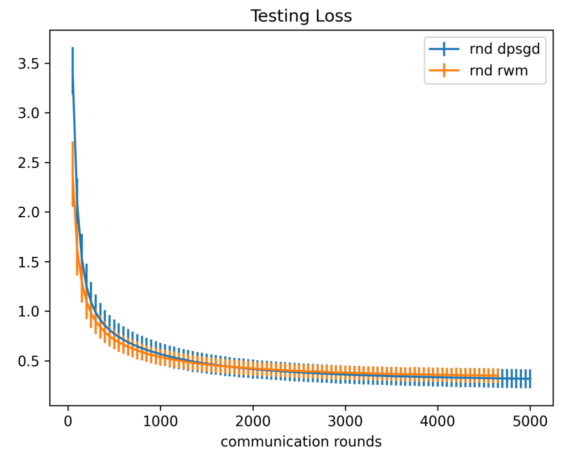

Figure 4.11: MovieLens: Test loss comparison between RWM and D-PSGD for random $\alpha$ sharing.

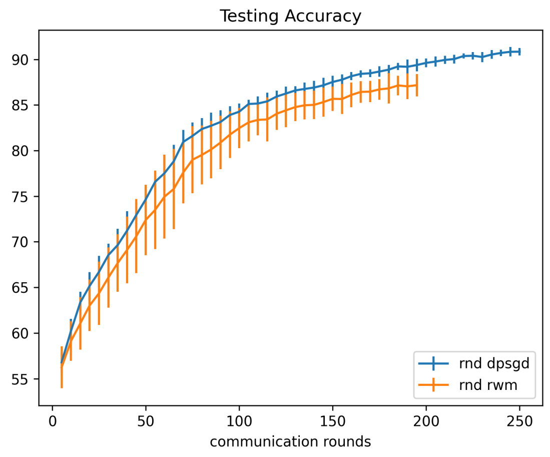

Figure 4.12: CelebA: Test accuracy comparison between RWM and D-PSGD for random $\alpha$ sharing.

### 4.2.4 Sub-Sampling Model

In this experiment, we examine the sub-sampling sharing on all three datasets configured in Table 4.1, performing local training on mini-batches. The choices of $\alpha$ also stay the same; those are 0.1, 0.3, and 0.5. For all datasets, the model performance with both RWM and D-PSGD algorithms is correlated to the fraction of model sharing. The differences are not that extensive, but the models give the best results when sending 50% of the model, slightly worse when sending 30%, and the poorest with 10% sharing. This behavior can be explained by the fact that sub-sampling makes a randomized choice of which parameters to share, unlike partial sharing, which takes the ones that have changed the most since the last time they were shared. Furthermore, sub-sampling performs worse than sending an entire model in each round.

The behavior is the same for the recommender dataset MovieLens when sharing the whole model, except that performance is worse. The RWM initially learns more in the first iterations, but D-PSGD improves a bit later and stays better. The test loss in $390^{th}$ iteration is 1.0501 for RWM and 1.0002 for D-PSGD, where the latter model terminates after 500 iterations with 0.8878 loss. Similarly, the CelebA dataset used for classification gives a clearer pattern of $\alpha = 0.5$ performing the best, again with identical behavior as with its full model sharing, only worst performance.

The experiments for FEMNIST dataset show similar patterns as the previous two described. The differences between various models are small, both for RWM and D-PSGD. In Figure 4.13 we plot test accuracy for RWM, while Figure 4.14 represents that for D-PSGD. Both communication techniques give slightly better results when randomly sharing 50% of the model parameters. Where in Figure 4.15, those best results are compared after the system executes 300 iterations. Here, the accuracy does not have an obvious pattern, as RWM performs slightly better in the first iterations, while after, the D-PSGD takes its place. The accuracy for terminating iterations of RWM and D-PSGD is 77.98% and 81.03%, respectively.

The time taken for RWM to complete its run was 144.8 minutes, while D-PSGD took 197.08 minutes in total. In Figure 4.16, we plot previously described results taking the time execution of RWM. This does not say much, as the results are similar to before, meaning that the time execution here does not impact much, and there is no trade-off. On the contrary, there is an important trade-off regarding the network load and model accuracy. As D-PSGD has much more communication and model exchanges in the network, in Figure 4.17, we show that load by comparing it to RWM on the average amount of data shared per node. If we estimated the total network load in both systems runs and limited them to use the same amount of data, we would obtain results depicted in Figure 4.18. It can be observed that there is a significant network load trade-off with regards to model accuracy, as the D-PSGD would achieve accuracy slightly bigger than 46.85%, while RWM would be much better with acquiring 77.98%.

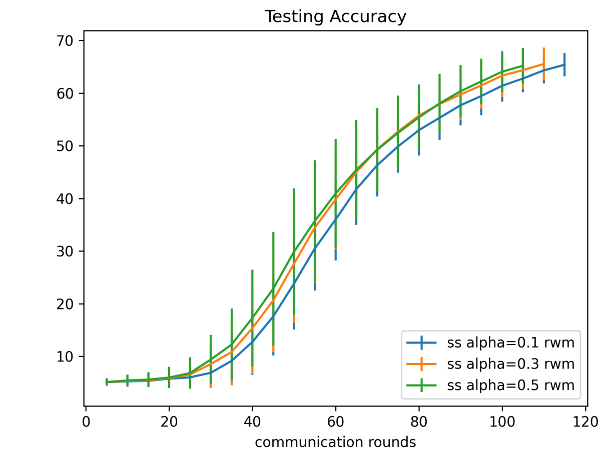

Figure 4.13: FEMNIST: Test accuracy for sub-sampling sharing with RWM.

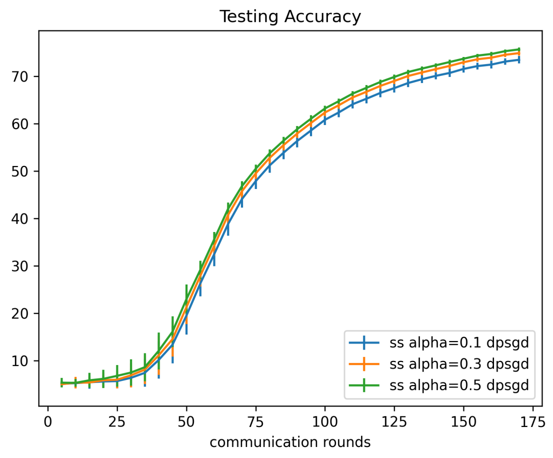

Figure 4.14: FEMNIST: Test accuracy for sub-sampling sharing with D-PSGD.

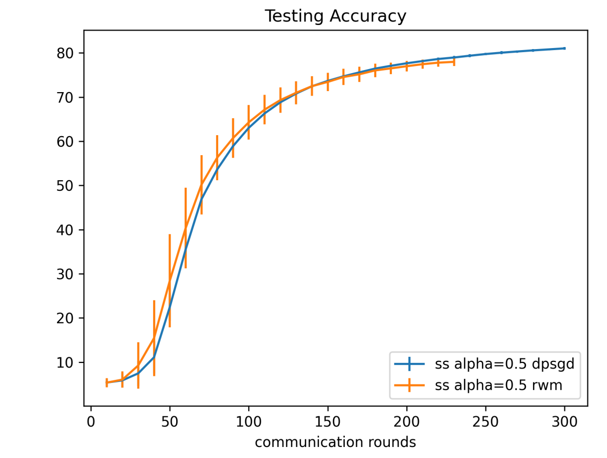

Figure 4.15: FEMNIST: Test accuracy comparison between RWM and D-PSGD for $\alpha = 0.5$.

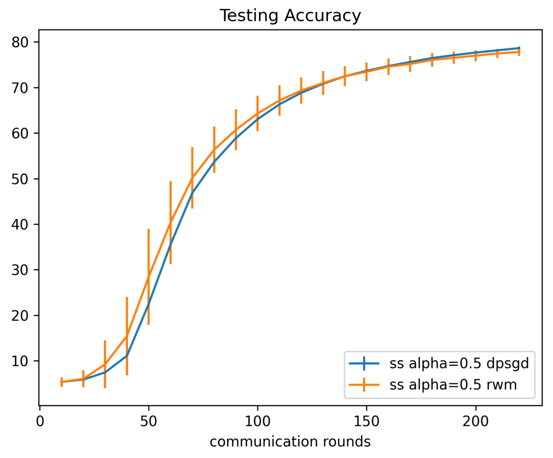

Figure 4.16: FEMNIST: Test accuracy comparison given the same processing time.

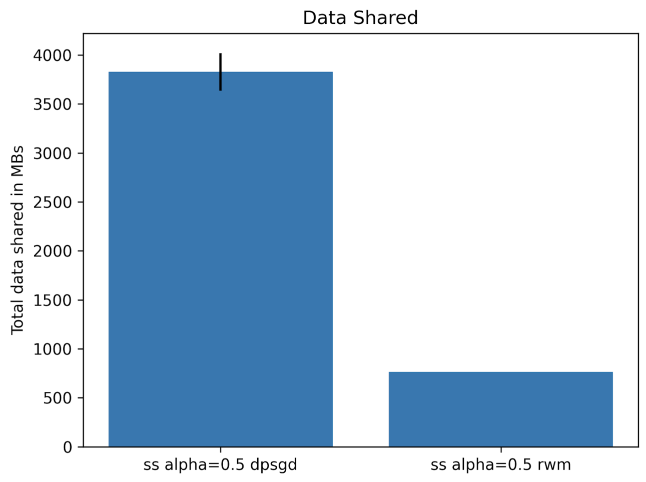

Figure 4.17: Average amount of data shared per node in D-PSGD and RWM.

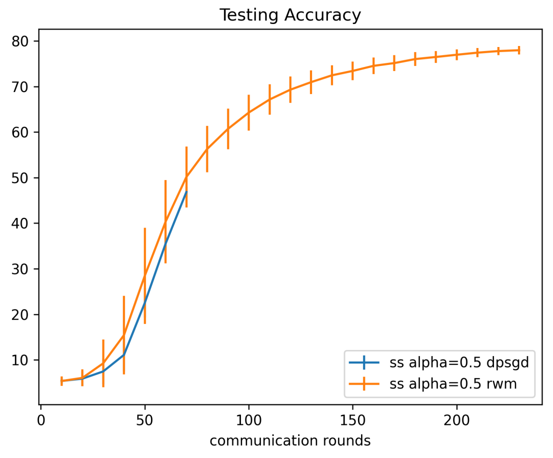

Figure 4.18: FEMNIST: Test accuracy comparison given the same network load.

# Chapter 5

# Conclusion

In this project, we examined the performance of a decentralized machine learning system when put through different communication techniques and various model-sharing methods. We observed the performance of D-PSGD and RWM algorithms used for communication and local model updates executed on three different datasets. Those datasets, namely MovieLens Latest, CelebA, and FEMNIST, cover both recommendation and classification systems, where machine learning algorithms used are matrix factorization and convolutional neural networks. Additionally, which model parameters the nodes exchanged among themselves and how much is varied through a couple of sharing methods.

We observed how usually sharing model parameters with all the neighbors and updating the local model using Metropolis-Hastings weight averaging, i.e., using D-PSGD, yields better results. When experimenting with a recommendation dataset, we conclude that the differences between the two approaches are almost negligible. However, the same can not be said for the image classification dataset CelebA, which consistently produced a better performance using the D-PSGD model. For the FEMNIST dataset, the line is blurry, but the D-PSGD still outperforms. Hence, the impact can also be found in model size and its nature. All this is concluded based on machine learning model accuracy or evaluation loss. Furthermore, the results remain the same when measuring the total processing time a run takes, meaning that time does not represent a significant burden. In contrast, we have an important trade-off for the two algorithms in terms of network load. As the RWM has less communication and computations, the amount of data exchanged is considerably less. Hence, a random walk model would produce a far better model if we were to limit the system's network traffic.

## 5.1 Future Work

This research can be further extended; hence, we present some ideas for future implementations. One direction of development lies in coming up with a different approach to sending information and updating the model in the RWM algorithm. For instance, a node can periodically send its model parameters to a randomly chosen neighbor, which would increase the pace of node advancement to further rounds and possibly lead to better performance. Additional work can be applied to mixing the two communication techniques to achieve the best possible performance without overloading the network. Since we have observed in a few experiments patterns that indicate better RWM performance only at the start of a run, it might be useful to create a system that would behave as an RWM at the beginning and then change to using the D-PSGD approach. Also, there should be more research regarding the performance of the methods on different network topologies. Since one of the algorithms is randomized, the performance might significantly differ based on the number of nodes in the network and how they are connected.

# Bibliography

[1] Towards Data Science. S. Saha. Dec 15, 2018. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.*
`https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

[2] Grouplens. 2021. MovieLens Latest Datasets.
`https://grouplens.org/datasets/movielens/latest/`

[3] arXiv:1812.01097. December 9, 2019. *LEAF: A Benchmark for Federated Settings.*
`https://arxiv.org/abs/1812.01097`