Implementation and convergence comparison of decentralized deep learning algorithms

by Aman Bansal

Project Report

Professor Anne-Marie Kermarrec
Project Advisor

Dr. Erick Lavoie and Dr. Rafael Pires
Project Supervisors

EPFL IC IINFCOM SaCS
BC Building, Office BC 347
CH-1015 Lausanne
Switzerland

January 15, 2021
Chapter 1

Introduction

Federated Learning (FL) involves training a global machine learning model on data stored across devices without the data ever leaving the device. This method has become an important area of research due to the rising concerns over data privacy and the success of machine learning. Decentralized FL (DFL) algorithms aim to enable such training in settings where the devices might not be fully connected, such as over a P2P architecture. Furthermore, it aims to speed up the training process by removing the need for a server, a communication bottleneck, while not sacrificing on convergence guarantees. These are gossip algorithms as each device only communicates with its neighbors.

The problem we wish to consider is that of obtaining a model representative of the data across all devices, on all devices. We focus on two such algorithms, Decentralized Parallel Stochastic Gradient Descent (D-PSGD) [1] and Random Model Walk (RMW) [2]. These algorithms work using the same principle: nodes alternating between training their models on local data and combining their models with those of their neighbors.

In this project, we develop an API using PyTorch [3] that abstracts away the implementation details of these algorithms and lets the user focus on the machine learning aspect. Furthermore, since RMW uses less network bandwidth than D-PSGD by design, we wish to empirically evaluate whether models trained with RMW can achieve similar accuracies as those trained with D-PSGD in the same number of epochs of training. We do this by using our API to train models for the MNIST and CIFAR-10 image classification problems. We finally report the results of our experiments and try to understand the effect of the network topology and other factors on the performance of these algorithms.
Chapter 2

Background

In this chapter, we present the central theme of this project, namely decentralized stochastic gradient descent and introduce the terminology used in the rest of the text. We then give a brief overview of the two algorithms that we implement and compare.

2.1 Stochastic Gradient Descent

Given a function \( l : \mathbb{R}^n \rightarrow \mathbb{R} \), typically some kind of loss function, \textit{gradient descent} attempts to solve the optimization problem

\[
\min_{x \in \mathbb{R}^n} l(x)
\]

by updating the parameters \( x \) as follows:

\[
x \leftarrow x - \gamma \Delta f(x)
\]

where \( \gamma \) is the \textit{learning rate}. The gradient here is computed on the entire dataset.

In \textit{Stochastic} gradient descent (SGD), the gradient is instead computed and the parameters updated one-by-one for each data point. In addition, the dataset is shuffled once we have gone through all data points. SGD generally performs better at avoiding local minima than simple gradient descent.

\textit{Mini-batch SGD} makes an additional optimization by taking a batch of data points instead of just one. This lets computers (especially GPUs) to make use of vectorization libraries and perform the step faster. In the rest of this text, we omit "Mini-batch" and simply use SGD to mean mini-batch SGD with a configurable batch size.
2.2 Decentralized Stochastic Gradient Descent

Decentralized SGD is the application of the above technique in a decentralized setting. The variant we consider here consists of achieving a set of parameters similar to that achieved by performing SGD on all the data as discussed above. We only consider the case where each node's data is independently and identically distributed (i.i.d.). The main constraints in such a setting are that a node only has access to its local data and can only communicate its model weights to its (possibly very few neighbors). The only requirement on the network topology is that all nodes be connected, since there is no way to get any information from a disconnected node's data.

We focus on two algorithms for this optimization problem. Here we only give a high-level overview of these algorithms as implementation details are provided in the next chapter.

2.2.1 Decentralized Parallel Stochastic Gradient Descent (D-PSGD)

In the D-PSGD algorithm, each node averages its local model with those of all its neighbors after each step of training on local data.

**Algorithm 1** Decentralized Parallel Stochastic Gradient Descent (on one node)

1: Initialize local model
2: while true do
3:   Compute the neighborhood weighted average by exchanging models with neighbors.
4:   local model ← average
5:   Perform a training step on the local model
6: end while

2.2.2 Random Modem Walk (RMW)

In the RMW algorithm, each node periodically sends its local model to one of its neighbors, selected uniformly at random. On receiving a model, it updates its local model using an update strategy. In our implementation, this update is done by averaging the incoming and the local model and performing a training step on the averaged model.

**Algorithm 2** Random Model Walk (on one node)

1: Initialize local model
2: while true do
3:   wait(ΔT)
4:   Send local model to a neighbor selected uniformly at random.
5: end while
6: 
7: procedure ON RECEIVED MODEL(m)
8:   Update local model by averaging with m
9:   Perform a training step on the averaged model
10: end procedure
Chapter 3

Implementation

One of the main results of this project is an API\(^1\) which enables the user to easily use the decentralized deep learning algorithms on a problem of their choice. In this section, we first document the usage of the API and then provide the details of the software architecture.

3.1 The Application Programming Interface (API)

This API is built in Python 3 and consists of 2 (abstract) classes which are to be inherited by a concrete implementation. We describe the role of these classes and their methods below.

Note that this API requires the \textit{torch}, \textit{torchvision}, \textit{pyzmq}, and \textit{numpy} packages.

Example uses of the API on the MNIST and CIFAR-10 classification problems can also be found in the repository.

3.1.1 LocalMain

The objective of LocalMain is to define all necessary operations to be performed on any node \textit{locally}. The constructor of the class is called with at least the rank of the node. However, the user can easily add any further parameters to be passed to their implementation of LocalMain. For instance, as all nodes had access to all data in our experiments (this is not a requirement of the API!), we also passed the total number of nodes in the network to this constructor to enable the nodes to sample disjoint data points. Another example of such a parameter could be the learning rate to be used in the training.

In addition to the constructor, the user also needs to define a \textit{step} function which performs one local training step on the node’s latest model on one mini-batch of data. This step typically consists of passing data through the model, calculating the gradient, back-propagation, and the optimizer step.

\(^1\)https://gitlab.epfl.ch/sacs/distributed-ml-api/api
3.1.2 Network

The objective of the Network class is to define the network topology, the details of the communication protocol, and the encoding/encryption to be used. The user needs to implement the following methods which completely define this class.

- `addr(rank, machine_id)`
  gives the connection endpoint for a node (see 3.2.1)

- `neighbors(rank, machine_id, procs_per_machine, machines)`
  gives the set of neighbors (rank, machine_id tuples) for any neighbor

- `encrypt(s: str)`
  all outgoing messages pass through this method, meaning that the communication can be easily encrypted using any protocol used in this method. By default, no encryption is used.

- `decrypt(b: bytes)`
  the counterpart of the encrypt method above.

3.2 Software Architecture

In this section, we describe the implementation details of the API. We discuss how we achieved the basic peer-to-peer communication and talk about the implementations of the two algorithms.

3.2.1 Communication - ZeroMQ

We experimented with two communication patterns from the ZeroMQ library: Router-Router and Router-Dealer.

In the Router-Router pattern, each node starts one Router socket and listens for connections and sends connection requests to the neighbors it wishes to connect to. However, the problem with this pattern is the way Router sockets work. A Router socket only initializes a message queue after a connection is established. This means that a router's connection request is only sent once and if the other side hasn't been initialized yet, the request is lost silently. This leads to the crux of the problem that is a Router A can only talk to another Router B once B has talked to A, which leads to a deadlock.

In the Router-Dealer pattern, in addition to a Router socket, a node creates as many Dealer sockets as it has neighbors. The advantage of this is that a Dealer socket creates its message queue as soon as it is created and periodically tries to deliver its message (automatic retry). We do not have to worry about trying to resend lost messages since this type of socket takes care of that. Hence, this pattern is reliable regardless of initialisation order of the nodes. The downside of this pattern, however, is the creation of one TCP connection for each messaging direction. This leads to doubling the required number of connection since TCP allows two-way communication with a single connection. Still, with the small number of neighbours of most nodes, the
additional file handlers used by networking libraries for book-keeping are outweighed by the convenience and reliability of using ZMQ’s Dealer sockets to abstract away the automatic retry mechanism.

The Router-Dealer pattern we used for true-peer connectivity (where each node is equal) is known as the Harmony pattern in the ZMQ guide\(^2\), where a more thorough discussion on it can be found.

### 3.2.2 Training Step Implementation

Here is the implementation of a local training step, used as a procedure in the algorithms that follow.

<table>
<thead>
<tr>
<th>Algorithm 3 Training step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( X, y \leftarrow ) next batch of samples, ground truth-labels</td>
</tr>
<tr>
<td>2: ( \text{output} \leftarrow \text{model}(X) ) (\triangleright) Feed-forward, (\text{output}) holds the predicted labels</td>
</tr>
<tr>
<td>3: ( \text{loss} \leftarrow \text{negative}_\text{log}_\text{likelihood}(\text{output}, y) ) (\triangleright) Compute loss for each prediction</td>
</tr>
<tr>
<td>4: ( \text{loss.backward}() ) (\triangleright) Back-propagation</td>
</tr>
<tr>
<td>5: ( \text{model.weights} \leftarrow \text{model.weights} - \gamma \ast \text{model.gradient} ) (\triangleright) Gradient Descent</td>
</tr>
</tbody>
</table>

A detailed explanation of how the `backward` function works can be found in this PyTorch tutorial\(^3\). To summarize, the backward function performs the back-propagation using the loss function starting at the output of the neural network. This call accumulates the gradient for each loss value, and hence each sample, in the gradient of the network. Finally, the gradient descent step updates the network weights which is how the network "learns" from the batch of data.

---

\(^2\)https://zguide.zeromq.org/docs/chapter8/#True-Peer-Connectivity-Harmony-Pattern  
\(^3\)https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
Each node runs the following algorithm:

**Algorithm 4** Decentralized Parallel Stochastic Gradient Descent (D-PSGD) implementation

1: Setup sockets, initialize local model $l$, load data (and choose a subset based on node's rank)
2: $\forall$ neighbor $i$, initialize an empty queue $q_i$
3:
4: **Send** local model and degree to all neighbors
5:
6: **Wait** for disconnection probes from all neighbors
7: Close all sockets and exit
8:
9: **procedure** **ON** **RECEIVE** **MODEL**($m$, $n$): $\triangleright$ where $m$ is the model, $n$ is the neighbor
10: $\text{Push } m \text{ to } q_n$
11: if $\forall q_i$ is non-empty then
12: \hspace{1em} $m_i := \text{Pop } q_i$
13: \hspace{1em} avg $\leftarrow$ compute average using Metropolis Hastings $\triangleright$ explanation below
14: \hspace{1em} local model $\leftarrow$ avg
15: Perform local **training step** (3.2.2)
16: **Send** local model and degree to all neighbors
17: if required number of epochs reached then
18: \hspace{1em} **Send** disconnection probes to all neighbors
19: end if
20: end if
21: **end procedure**

Note: We could remove the nodes sending their degrees every iteration since we are using a fixed topology. However, sending it every iteration allows for a flexible topology, something that we experimented with at one point but is not part of the report.

The weights attributed to each model received by a node are calculated using the Metropolis-Hastings protocol [4]. This calculation can be summarized as follows:

$$W_{ij} = \begin{cases} \frac{1}{\max(d_i,d_j)+1} & i \neq j, i \leftrightarrow j \\ 1 - \sum_{j \in N_i} \frac{1}{\max(d_i,d_j)+1} & i = j \\ 0 & \text{otherwise} \end{cases}$$

where $N_i$ is the set of neighbors of $i$. $i \leftrightarrow j$ denotes that $i$ and $j$ are connected.

Due to the nature of the algorithm itself, the nodes move in lockstep, i.e., remain synchronized in terms of the number of training steps completed. Thus, we can easily establish a proper ending strategy as all nodes reach the required number of epochs after the same number of iterations.
3.2.4 RMW

Each node runs the following algorithm:

Algorithm 5 Random Model Walk (RMW) implementation (on one node)

1: Setup sockets, initialize local model $l$, load data (and choose a subset based on node’s rank)
2: $\forall$ neighbor $i$, initialize an empty queue $q_i$
3:
4: **Send** connection probes to all neighbors
5: **Wait** for connection probes from all neighbors
6:
7: **for** number of epochs $\times$ number of mini-batches **do**
8: \hspace{1em} **Wait** $\Delta$ \hspace{1em} \text{\textcopyright Discussion on } \Delta \text{ below}
9: \hspace{1em} **Send** local model to a neighbor chosen uniformly at random
10: **end for**
11:
12: **Send** disconnection probes to all neighbors
13: **Wait** for disconnection probes from all neighbors
14: Close all sockets and exit
15:
16: **procedure** ON Receive MODEL ($m$)
17: \hspace{1em} $l \leftarrow \frac{l + m}{2}$
18: \hspace{1em} Perform local training step (3.2.2)
19: **end procedure**

The choice of $\Delta$ here is crucial as we explain below.

Indeed, if we choose a value too low, the nodes will send and receive lesser-trained models which will harm the training at all nodes. For instance, consider the extreme case where $\Delta$ is very small such that all nodes send their untrained models $\#epochs \times \#mini-batches$ times before having the time to receive any models and train on its data. They will eventually receive models and will, every time, average with them and train on the averaged model. This will result in worse performance than if the node had access to only its data. Firstly, the arriving models are bringing no information from other nodes’ data. In addition, the trained model is being averaged with arriving models which were randomly initialized, thus the nodes are losing information of their own data!

We can conclude that our choice of $\Delta$ has to be large enough, since making it too big will not hurt training. In fact, the optimal value for training would be bigger than the (training + averaging times) $\times$ (degree of a node) because, in theory, a node could receive one model from all its neighbors during the same iteration. An upper bound on this is indeed what we use for final experiments. However, larger times obviously slow down training, which brings to a trade-off between the total training time and the convergence rate. Still, one can choose an optimal $\Delta$ which optimizes this trade-off in the average case. For regular topologies where all nodes have the same degree, we expect (in the mathematical sense of the term) a node to receive one model per iteration. Assuming that all nodes have identical performance and network latency, the optimal $\Delta$ can be chosen to be the sum of the average latencies of the training, averaging, and transfer steps.
3.2.5 RMW-Async

To remove the need to tune the $\Delta$ parameter, we evaluate a slight variation we called RMW-Async along with RMW. In this version, instead of the nodes sending their models every $\Delta$ time, they only do so when they receive a model. A study of equivalence between the two algorithms is left for future work.

**Algorithm 6** RMW-Async implementation (on one node)

1: Setup sockets, initialize local model $l$, load data (and choose a subset based on node’s rank)
2: $\forall$ neighbor $i$, initialize an empty queue $q_i$
3: 
4: **Send** local model to a neighbor chosen uniformly at random.
5: 
6: **Wait** for *disconnection probes* from all neighbors
7: Close all sockets and exit
8: 
9: **procedure** **ON** **RECEIVE** **MODEL**(m)
10: $l \leftarrow \frac{l + m}{2}$
11: Perform local **training step** (3.2.2)
12: if number of epochs reached **then**
13: **Send** disconnection probes to all neighbors
14: else
15: **Send** local model to a neighbor chosen uniformly at random
16: **end if**
17: **end procedure**

3.2.6 Implementation Notes

To implement these algorithms without busy waiting and concurrently (especially RMW as it requires two simultaneous active procedures for sending and receiving models), we use the asyncio library and its event loop. In particular, the gather method allows us to add the functions we wish to run concurrently to the event loop and let asyncio take care of the scheduling.

To actually train using a given topology and dataset, we use one process for each node in the topology where each of them runs the same procedure. These processes can be on the same machine or on different machines. PyTorch’s multiprocessing library allows us to easily spawn as many processes as nodes, as done in the example implementations of MNIST and CIFAR-10.
Chapter 4

Evaluation

4.1 Methodology

4.1.1 Convergence Analysis

To evaluate the performance, we log the model at each node every epoch, as defined below for each algorithm. After the training, we run a testing script which reads this model and uses the test dataset to evaluate the accuracy of the logged models. To counter inhibitingly long testing times, this script tests the models of the different nodes in parallel and speeds up this part of the experiments by a factor of min(#Nodes, #CPUs).

We define an epoch locally for a node as follows for each algorithm:

- **D-PSGD**: sampling through the entire data-set. Since nodes progress in lockstep, they remain synchronized in the number of epochs.
- **RMW**: sending as many models as there are mini-batches of data.
- **RMW-Async**: sampling through the entire data-set.

In RMW-Async, one problem is the determination of a fair finishing strategy for RMW-Async. This is because this algorithm doesn't guarantee that all nodes will progress synchronously (in the number of epochs). We considered two possibilities:

- Training ends when all nodes have reached the specified number of epochs or more: The problem with this strategy is that the nodes which have reached a higher number of epochs will communicate their models to others. This, in turn, implies that even if we discard the results after a certain epoch, the nodes may have benefited from "over-trained" models, which would result in an unfair evaluation.
- Training ends when one node has reached the specified number of epochs: This strategy, being the other extreme, gives a disadvantage to RMW-Async as most nodes will not have trained until the specified number of epochs when training ends.
We chose the second strategy, as it results in a stricter evaluation of RMW-Async. This is because choosing the lenient strategy might be misleading, especially since the convergence of RMW-Async hasn’t been studied as opposed to the other algorithms.

4.1.2 Datasets and Models

We use the MNIST and CIFAR-10 datasets to test our implementation and evaluate the algorithms.

MNIST

The *MNIST* dataset consists of 28 * 28 pixel gray-scale images of handwritten digits (from 0 to 9) along with their ground-truth labels. The training dataset contains 60,000 images while the test dataset contains 10,000 images. We consider the simplest model for MNIST, a fully connected neural network with 7850 trainable parameters, as shown in Figure 4.1.

![Figure 4.1 – The fully connected network for MNIST. Instead of the 28x28 input nodes on the left, only 16 are shown](image)
CIFAR-10

The CIFAR-10 dataset consists of 32 * 32 pixel color images of 10 different classes - airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks - along with their ground-truth labels. The training and test datasets contains 50,000 and 10,000 images respectively. We consider a variant of LeNet, with 85354 trainable parameters, as shown in Figure 4.2.

The exact parameters of the convolutional network are the following:

```python
features = Sequential(
    Conv2d(3, 32, kernel_size=5, stride=1, padding=2),
    MaxPool2d(kernel_size=3, stride=2),
    GroupNorm(2, 32),
    ReLU(inplace=True),
    Conv2d(32, 32, kernel_size=5, stride=1, padding=2),
    GroupNorm(2, 32),
    MaxPool2d(kernel_size=3, stride=2),
    ReLU(inplace=True),
    Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
    GroupNorm(2, 64),
    MaxPool2d(kernel_size=3, stride=2),
    ReLU(inplace=True),
)
classifier = Linear(576, 10)
```

4.1.3 Topologies

Here we present the example topologies used in our experiments. The range of topologies, in terms of connectivity, is from fully connected to barely connected (ring). This lets us analyze both algorithms in multiple ways and evaluate how they perform under different conditions.
**Fully Connected**

Every node is connected to all other nodes in the network.

![Figure 4.3 – A fully connected topology with 6 nodes](image)

**Ring**

Every node is connected to its immediate rank neighbors in the network. A node with rank r is connected to the nodes with ranks:

- \((r - 1) \text{ mod } \text{world}_\text{size}\)
- \((r + 1) \text{ mod } \text{world}_\text{size}\)

![Figure 4.4 – A ring topology with 6 nodes](image)

**Grid**

Every node is connected to its spatial neighbors when the nodes are arranged in a square grid.

![Figure 4.5 – A grid topology with 9 nodes](image)
4.1.4 Test Environment Specifications

All experiments were conducted on the labostrex128-130 machines (in the IC cluster at EPFL). They all have the same specifications: 32 CPUs, each of which is an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, with 125GB of RAM.

4.2 Micro-benchmarks: System Costs of Basic Operations

In this section, we present various micro-benchmarks which help us profile these algorithms and to identify likely bottlenecks during execution. The micro-benchmarks were chosen to reflect the main operations performed by both algorithms: transferring models over the network, performing a local training step, and averaging models, for both MNIST and CIFAR-10. For both models, we found that the transferring step largely dominates the other two steps, by an order of magnitude. Furthermore, while the averaging step's latency is overshadowed by that of the training step's for CIFAR-10, both are of the same order for MNIST. The results were also used to tune the $\Delta$ in RMW as discussed in 3.2.4. More detail on each experiment are provided in the next sections.

4.2.1 Transferring Model Weights

We measure the latency of transferring models between processes as this is a potential candidate for a bottleneck in the complete training process. Concretely, these are the steps we wish to consider:

- Node A: Serialize model into a json string
- Node A: Encode model into bytes
- Node A: Send model to another process (same or different machine)
- Node B: Receive model
- Node B: Decode model from bytes
- Node B: Deserialize model from the json string

Since we cannot measure the one way (from node A to node B) time, we measure the time taken to perform these steps twice - back and forth between two processes. Figure 4.6 shows the average times measured with 500 iterations along with the standard deviations.
Figure 4.6 – Average round-trip times with standard deviations for encoding, transferring and decoding a model between two processes

Firstly, the latency of transferring is higher between machines than on the same machine for both datasets, as one would expect. The bigger increase (from same machine to different machines) for MNIST than CIFAR-10 could be because of the overhead of transmitting through the network, which becomes less important with more data (larger model).

We also note that the times for CIFAR-10 are an order of magnitude bigger than those of MNIST, consistent with difference in the size of their models, also an order of magnitude.
4.2.2 Training Step

Figure 4.8 shows the time taken to perform one step of training with different batch sizes, averaged over 100 repetitions.

![Bar chart](image)

Figure 4.7 – Average time taken to perform a training step, divided into its 3 constituent phases with standard deviation. The scale used for batch sizes is logarithmic.

We measure the three main phases of a training step to identify those that are dependent or independent of the batch size.

- **Feed Forward** - Passing all samples in a batch through the neural network.
- **Back Propagation** - Computing the gradient for each model weight based on each sample.
- **Gradient Descent** - Updating model weights using the gradient.

The feed forward and back propagation phases are dependent upon the batch size as they repeat the same process for each sample in a batch. The gradient descent phase is independent of the batch size as it simply updates the model weights by subtracting the average gradient due to all samples.

Firstly, we note that the average total time taken for one step is between one and two orders of magnitude larger for CIFAR-10 than MNIST, in general. For example, with a batch size of 128 that we used in our final experiments, the time for MNIST is about 0.5ms while it is 25ms (50x). This can be explained by the fact that the LeNet used for CIFAR-10 has around 11x more trainable parameters (85354 vs 7850) and that each sample (image) has around 4x more pixels (32*32*3 vs 28*28) than MNIST. The former increases the latency of all the three phases while the later increases the latency of the feed forward phase.
A second observation is that difference in the relationship between batch size and latency for MNIST and CIFAR-10. The growth of the latency for CIFAR-10 can be seen to be linear as it is almost doubled with each doubling of the batch size. For MNIST, on the other hand, this not the case and the growth is sub linear. One possible explanation is that since MNIST’s latency is much lower in absolute numbers and since its LinearNet is a very simple fully connected layer, the software overhead is more important at this scale than the growth due to the batch size. Another possibility is cache optimizations which might be possible for MNIST given it’s relatively low model size and not for CIFAR-10’s LeNet.

### 4.2.3 Averaging Models

Here we measure the time taken to average a varying number of models of each type (MNIST LinearNet and CIFAR-10 LeNet), each experiment being repeated 100 times.

![Figure 4.8 – Time taken to average models](image)

A linear relationship between the number of models and the average latency is clear from Figure 4.8. We can also see that, for a given number of models, the averaging time for CIFAR-10’s LeNet is about an order of magnitude larger than that of MNIST’s LinearNet, similar to the difference between their number of parameters.

### 4.2.4 Conclusion

We consider a network with 25 nodes on the same machine and a batch size of 128. The number of models being averaged in RMW is always 2, while in D-PSGD, this number is one more than the degree of the node.
For MNIST, the averaging latency cannot be neglected with respect to the training time, even for RMW since averaging two models takes about $0.1ms$ which is a fifth of the training time of $0.5ms$. For D-PSGD, especially with a fully connected topology on 25 nodes, the averaging latency becomes about 3 times that of training, namely $1.5ms$ vs $0.5ms$.

For CIFAR-10, since the averaging latency for 2 models is about $1ms$, it can always be ignored for RMW since the training latency is about $25ms$ with a batch size of 128. We can also neglect it with D-PSGD and the ring and grid topologies as the averaging latency remains lower than 10x the training latency. However, with D-PSGD and the fully connected topology, the averaging time of $12.5ms$ is significant compared to the training time.

Finally, the model transfer time dominates both the other phases, being an order of magnitude higher than the training and averaging times. Hence, it is the obvious bottleneck for all algorithms. Even though RMW sends and receives less models per iteration than D-PSGD, the latency should stay about the same since at least one model is being sent/received. RMW will have a latency advantage over D-PSGD in a situation where the network bandwidth is low enough such that sending/receiving more models increases the latency of this step significantly.

For RMW, we choose the $\Delta$ parameter to be more than sufficient as described in 3.2.4, to make sure that various factors such as OS scheduling do not influence our results. For MNIST, the sum of the average latencies of the three steps is roughly $20.7ms$ ($20ms$ transfer on the same machine, $0.5ms$ training, and $0.2ms$ averaging) and we used $\Delta = 500ms$. For CIFAR-10, this sum is $307ms$ ($280ms$ transfer on the same machine, $25ms$ training, and $2ms$ averaging) and we used $\Delta = 3000ms$.

### 4.3 Comparison of D-PSGD, RMW, and RMW-Async

In this chapter, we present the results of the algorithms on the MNIST and CIFAR-10 classification problems. Following is a description of the experiments we use for our evaluation. **Experiment:** Train a classifier distributively on 25 nodes with an independently and identically distributed (i.i.d.) data distribution for 250 epochs with a batch size of 128 samples. We run the experiment for each of the three topologies presented before: ring, grid, and fully connected. We perform each of the above with D-PSGD, RMW, and RMW-Async.

We ran the experiments using different learning rates $\{0.001, 0.005, 0.01, 0.05, 0.1, 0.5\}$ and found $0.5$ and $0.1$ to be the best for MNIST and CIFAR-10 respectively, in terms of the average accuracy reached after 250 epochs. The results for these learning rates are summarized in the graphs in the following sections. For both datasets, there is one graph per topology where the performance of each algorithm is shown. The solid line indicates the average of the accuracies of the different nodes, while the faded error bars represent the standard deviation of the accuracies of the different nodes.

Note: Since all nodes don't reach 250 epochs in RMW-Async, the last accuracy reached by each node is duplicated to have 250 data-points per node. This allows us to compute the averages and standard deviations for each of the 250 epochs.
4.3.1 MNIST

Figure 4.9 – MNIST - 25 Nodes - Ring topology - Learning rate = 0.5
Average accuracy of nodes in bold with standard deviation shown as faded error bars
Figure 4.10 – MNIST - 25 Nodes - Grid topology - Learning rate = 0.5
Average accuracy of nodes in bold with standard deviation shown as faded error bars
Figures 4.9 and 4.10 show that all three algorithms achieve very similar performance throughout the training for the ring and grid topologies. For the fully connected topology, figure 4.11 shows that RMW-Async lags slightly behind D-PSGD and RMW-Async for the first 100 epochs or so. Thereafter, all three algorithms remain progress at a similar rate and achieve similar accuracies. We do note some divergence in the accuracies towards the end and D-PSGD seems to perform slightly better than the other two algorithms. These differences with the fully connected can be explained from the fact that the D-PSGD has the most advantage with this topology. This is because, in D-PSGD, each node averages its models with all its neighbors and the number of neighbors is the highest possible for this topology. This lets the nodes learn from everyone’s data at each iteration through the averaging step.

Nevertheless, the average accuracies for all algorithms for all topologies stay within their re-
pective standard deviations from one another. We can hence conclude that they have very similar convergence rates for the MNIST classification problem with the simple linear net. Now, we put these algorithms to test against a more complex model: CIFAR-10 with LeNet.

### 4.3.2 CIFAR-10

![Graph showing accuracy comparison between D-PSGD, RMW, and RMW-Async](image)

We observe from figure 4.12 that until around 200 epochs, D-PSGD performs better than RMW as the average accuracy is $1 - 3\%$ higher. We also note that the average accuracy of D-PSGD stays about the same after 150 epochs, meaning that it has almost reached its maximum. RMW catches up to D-PSGD after about 50 more epochs after that and stays constant thereafter. Hence, we can see
that D-PSGD does have a higher convergence rate in this case. Let us now consider the absolute bandwidth costs. Until convergence, each node in D-PSGD has sent $150 \times 2 = 300$ messages while each node in RMW has sent $200 \times 1 = 200$ messages. Hence, we can also conclude that RMW uses less bandwidth to reach the same accuracy.

We also note that RMW-Async performs slightly worse than RMW, especially after around 125 epochs. For this particular RMW-Async experiment, the minimum epoch reached by a node was 227. So, we can attribute the worse performance only after 227 epochs to the strict ending strategy used, the worse performance before must then come from the algorithm itself.

However, we should be careful and not draw strong conclusions from this experiment because the curves are within their respective standard deviations from one another.

Figure 4.13 – CIFAR-10 - 25 Nodes - Grid topology - Learning rate = 0.1
Average accuracy of nodes in bold with standard deviation shown as faded error bars
We note from figure 4.13 that D-PSGD converges faster than the other two algorithms. RMW averages do not lie within a standard deviation of D-PSGD averages, suggesting that there is significant difference in the convergence speeds and hence the final average accuracies. To explain this, we consider the key difference between the ring and grid topologies. Contrary to a ring, a grid's nodes do not all have the same number of neighbors. While implementing D-PSGD, we compensated for this fact by using the Metropolis-Hastings averaging protocol which takes into account the degree of the node and its neighbor to calculate the weight to be assigned to the corresponding incoming model. On the other hand, we did not introduce any such compensation in RMW as all averages are done pairwise where each model gets the same weight of 0.5. Metropolis-Hastings allows the information from the edge nodes to be well spread into the network. In RMW, on the other hand, the edge nodes remain unfairly represented and the inner nodes do not benefit from the information contained in the edge nodes' data. This effectively reduces the "mixing" between the nodes and the averaged model has little information about the data of edge nodes.

It is also interesting to note that RMW-Async actually converges faster than RMW. We explain this as follows: In RMW, all nodes send their models every iteration without regard to whether they have been trained since the previous iteration. On the contrary, in RMW nodes only send their model when they have performed another step of training. Since the edge nodes of the grid are less likely to receive models (as they have less neighbors), their models are likely to be less trained, for both RMW and RMW-Async. However, in RMW, these nodes will still send their models to a neighbor, which will be averaged with that neighbor's model. If this neighbor happens to be an inner node, its model will likely be more trained, meaning that the averaging step is likely to be detrimental to learning. In RMW, the edge nodes will only send a model if they received one in the previous iteration and trained upon it, thus supplying "newer, helpful information".

Furthermore, in this particular experiment for RMW-Async, the corner, edge, and inner nodes reached around 115, 180, and 240 epochs, respectively. This tells us that in an asymmetric topology like the grid, RMW-Async with the chosen strict ending strategy ends up learning less from the edge nodes' data. However, contrary to RMW, the outer lesser-trained models do not harm the inner higher-trained models, which is how we explain the better performance of RMW-Async, as compared to RMW.

To test our above hypothesis about compensating for edge nodes having less neighbors, we perform the same experiment with the Torus topology, where the edge nodes are connected to the opposite edge nodes, in a circular fashion. Hence, all nodes have the same number of neighbors, namely 4.
We see from figure 4.14 that all three algorithms converge at very similar rates for this topology. We still observe that RMW converges slower than D-PSGD until about 150 epochs. As compared to the results with a grid, the final average accuracies for RMW and RMW-Async have increased by $1 - 1.5\%$ while that of D-PSGD has decreased by about $3\%$. RMW and RMW-Async also reach higher average accuracies than D-PSGD after 250 epochs.
As seen in figure 4.15 for a fully connected topology, RMW and RMW-Async actually converge faster than D-PSGD and reach higher final average accuracies. This is quite an unexpected result, since D-PSGD with a fully connected topology is expected to be equivalent to local training on the combined datasets of all nodes.

If we consider the three regular topologies (ring, torus, and fully connected), we can make two observations. Firstly, D-PSGD’s performance compared to the other two algorithm decreases as the connectivity of the nodes increases. Secondly, the epoch to epoch variation (or noise, as some changes are positive while others are negative) in the accuracy of D-PSGD increases as the connectivity of the nodes increases. A possible explanation of these observations could come from the averaging step of these algorithms. As the connectivity of the nodes increase, the number of
models averaged at each iteration by D-PSGD increases, contrary to RMW and RMW-Async where it stays constant. Since the averaging and training steps alternate, we hypothesize that averaging more and more models does not work well with the training step and ends up introducing noise. However, it is important to note that since the data is distributed identically across nodes, the models should have similar weights, in theory, and hence averaging them should not "compete" with the training step. This unexpected behavior needs further investigation, perhaps using another dataset or training with a higher number of nodes, which unfortunately couldn't be a part of this project due to time constraints. Also, an adaptive learning rate is another direction to explore, as it might help in reducing this "competition" between the training and averaging steps and help the models move faster towards the global average.
Chapter 5

Conclusion

In this project, we implemented an API which facilitates the use of the decentralized deep learning algorithms D-PSGD and RMW using PyTorch and ZeroMQ. We also performed a convergence comparison of these algorithms on the MNIST and CIFAR-10 classification problems.

We observed that RMW converges slower compared to D-PSGD. Choosing a single random neighbour among a larger number of neighbours increases the time it takes for a node to learn from other nodes’ data, since there is less communication and averaging. We did see, however, that RMW could have an advantage over D-PSGD for applications where bandwidth is prohibitive or a bottleneck, since in RMW nodes send and receive less messages/models per iteration (by a factor equal to the degree of the node).

Another consideration with RMW was the tuning of its $\Delta$ parameter, which presented a trade-off between the training time and the convergence rate. While we chose to use a large $\Delta$ to focus on the convergence analysis, it could be interesting to optimize this choice and quantify this trade-off. To remove the need for this tuning, we also analyzed a variant of RMW referred to as RMW-Async where a node only trains and sends its model when it receives one. However, in our experiments with CIFAR-10, RMW-Async converged slower than RMW for the ring and fully connected topologies, suggesting that regularly sending models as RMW does might have an advantage.

RMW-Async did, however, converge faster than RMW with the grid. We hypothesized that its advantage with the grid comes from the fact that edge nodes do not send their lesser-trained models to others in RMW-Async, contrary to RMW. One direction for future work would be see if this can be remedied by increasing the likelihood of sending models to these edge nodes, by taking into account the degree of a node and its immediate neighbors.

Another direction of future work is to try to prove or disprove the convergence of RMW-Async. If RMW-Async does converge with the same theoretical bounds for some set of problems, it would be an improvement over the standard RMW as it removes the need to empirically choose the $\Delta$ parameter.
Bibliography


