Studying partially-centralized distributed learning

Master Project Report

by Mathis Randl

under the supervision of

Anne-Marie Kermarrec,

Martijn de Vos,

Rafael Pires,

Rishi Sharma,

Akash Dhasade

EPFL IC - SACS
BC 347 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 19, 2024

Acknowledgements

First of all, I am immensely grateful to all members of the SaCS team at EPFL for their immense help towards the accomplishment of this project. The initial ideas leading to the writing of this report were first imagined by them. The insights, guidance and companionship they provided made this project into what it currently is.

I am forever beholden to my family for all the love they gave, sacrifices they made, and patience they had to make me the person I am.

Perhaps on a more down-to-earth note, I would like to thank Infomaniak for providing us with free, powerful virtual machines that were a great help in reliability and experiment reproducibility. This project would have not been the same without them.

Finally, I also thank Mathias Payer and the HexHive lab more generally for providing the template for this report.

Lausanne, January 19, 2024 Mathis Randl

Abstract

Federated learning (FL) and decentralized learning (DL) are two of the main competitors in the space of distributed machine learning algorithms.

In this report, we investigate the viability of machine learning models trained partially using one algorithm and partially using the other, yielding unexpected insights into the relationship between both original algorithms.

We present how a change in the learning algorithm during the learning process empirically impacts the usual performance metrics of distributed learning, by running real-world ML pipelines on this hybrid setup and comparing the results against that of the stock implementations of FL and DL.

We then suggest ways in which these findings can be used to improve the current state-of-the-art algorithms with minimal modifications.

Contents

Chapter 1

Introduction

Machine learning (ML) has enjoyed large amounts of research and development over the last decade[7]. Most notably, the larger datasets made available and the vast increase in computing power have enabled significant progress in the feasibility and quality of various ML techniques. At first, typical ML pipelines only involved one computer, which would be tasked with some computation to carry out on an entire dataset. However, it later became apparent that such monolithic methods had innate problems, such as the computing power of a single computer not evolving as fast as hoped, or the more recent concerns about data privacy. This led to the development of distributed ML methods, which make use of several machines to reduce the amount of computing to be done on a single node, and provide ways to account for data privacy.

This decentralization of computing came with its own set of issues, as decentralized systems have well-studied limitations. In particular, it is not possible to detect from a worker node whether another node has failed, unless very specific assumptions are made about the underlying network. There are also legal constraints on the algorithms that may be deployed on such systems: some applications manipulate data so sensitive that it may not be shared from one node to the other. This means that only the worker node that holds a given data point is able to make meaningful, direct use of it. In this scenario, it may not even be the case that the data is independent and identically distributed across nodes.

Nonetheless, several distributed ML algorithms emerged that fit the constraints described above. Most notably, two competing architectures for distributed ML were developed with different assumptions:

Federated Learning (FL) [6] makes a difference between *worker* nodes, and a *server* node that is typically assumed to be perfectly available. Workers receive a model from the server, perform some work on it, and send their updated model back to the server. The server is tasked with the distribution of the current model, followed by the aggregation of the updated ones to form the next model to distribute. This cycle is repeated a large number of times or until satisfactory performance is reached. This approach mitigates the issue of data privacy by guaranteeing that no raw data ever leaves a given node, but the main issue arising in monolithic setups remains: a failure from a single node can prevent all progress, if that node happens to be the centralizing server.

Decentralized Learning (DL) [4] discards the idea of centralizing the models, by assigning to each node a set of *neighbors* (which are themselves other worker nodes). Here, each node trains its own model, shares it and obtains that of its neighbors, and merges them all into one. In a sense, each node behaves as a server for its neighborhood set. This method solves the issue of the availability requirements of any given node, as there is no single node whose failure would paralyze the whole system. In general, DL usually yields a slower convergence than that of FL, as nodes will only get updates from their closest neighbors and do not immediately benefit from the work of nodes that are further away in the neighborhood graph.

From the descriptions of both distributed learning algorithms we presented, we can notice a gap in their working hypotheses: we observe that the requirement that a server is always available can be relaxed into assuming that a server is occasionally available, but is not guaranteed to be. For example, the main server might irreversibly crash in the middle of a federated learning session, in which case one might want the workers to switch to DL for the remainder of the session. The opposite situation is also of interest, where one starts a DL session but wishes to possibly improve its results by performing FL iterations when the centralizing server later becomes available.

We therefore propose to study the behavior of such models in various settings that display their strengths and limitations. Since these models are trained in a way that is very similar to previously known distributed ML techniques, we will also study whether the results we obtain can be useful for improving the quality of ML pipelines that are currently used at large.

In this report, we analyze the performance of models trained partially in one algorithm and partially in the other. We explore the relationship between the model performance and the proportion of training rounds spent in one algorithm or the other. We also perform experiments to understand whether iterations that happen earlier in the learning process have a stronger impact on the overall learning quality than later ones. Finally, we also apply the results we obtain to improve the current state-of-the-art algorithms using very few modifications.

The research questions we ask are the following:

- **How does a change in the learning algorithm caused by a centralized server crash or recovery influence the quality of distributed learning?**
- **Is it possible to exploit these findings to improve the quality of distributed learning in any of the original algorithms?**

To the best of our knowledge, the idea of using both FL and DL algorithms on a single model is new, the experiments we describe have never been performed, and the results we obtain have never been reached.

Chapter 2

Background

In this section, we will provide a short introduction to several topics that we assume to be familiar to the reader. These revolve around machine learning, both on a single machine and distributed on a computing cluster. We also provide pointers to external resources that present these concepts in much greater depth.

2.1 Supervised learning with neural networks

Machine learning can mostly be split into three categories:

- Supervised learning[2], which deals with the problem of finding a relationship between some data points and their associated label, essentially learning to mimic an unknown function. When the labels are discrete, for example when training a model that differentiates between images of cats and dogs, the problem is called a "classification problem". Otherwise, i.e. when the labels are continuous, it is called a "regression problem".
- Unsupervised learning[1] deals with the problem of finding patterns in data which has no label, such as finding clusters of similar data points, or projecting them onto a lower-dimensional space while maintaining as much variance as possible.
- Reinforcement learning[5], which deals with systems that need to make decisions and are rewarded or punished by their environment.

In this report, we are exclusively interested in supervised learning.

2.1.1 Supervised learning

The typical problem encountered in supervised learning goes as follows: Suppose we are given an ordered list of x_i's, where $x_i \in \mathbb{R}^k$, as well as their associated $y_i \in E \subset \mathbb{R}$. We initially have no idea of the relationship between x_i and y_i.

We strive to find a deterministic function (a *model*) f so that $f(x_i) \approx y_i$ for every i. Ideally, this model generalizes correctly to unseen data, so that we have effectively found a reliable relationship between the input and the output data.

A good example of a simple supervised learning problem is ordinary least-squares regression, where we only consider functions of the form $f(x_i) = \sum_{j=1}^{k} w_j * (x_i)_j$. The problem is to find fitting w coefficients, so that f indeed approximates the relationship between x_i and y_i. We can compare how well two such coefficient vectors fit by preferring the one that minimizes the *least square loss* ($\sum_{j=1}^{n}(f(x_i) - y_i)^2$). A well-known result gives that the optimal weights can be computed analytically using the formula $w = (X^T * X)^{-1} * X^T * Y$, where $X_{i,j} = (x_i)_j$ and $Y_i = y_i$.

This specific problem shows the general pattern of supervised learning. We pick a family of models to consider, along with a loss function that enables the comparison of the performance of such models. We then pick the model that has the lowest loss on the given labeled data.

While this general pattern is simple to follow on paper, it has a major issue: it is very common that the family of models we initially considered as candidates is infinite (each w_i has an uncountably infinite range in OLS regression, for example). Minimizing the loss is therefore typically impossible to brute-force, and rarely has a closed-form solution as above. There are several ways to find a 'good-enough' model in this scenario, such as gradient descent or coordinate descent.

2.1.2 Neural networks

While OLS regression is very efficient to optimize, most problems do not display a simple linear relationship between the input and the output variables, which is a major assumption of linear regression techniques. Fortunately, another family of models has much better flexibility at the cost of having no known closed-form solution. Members of this family are called **neural networks**[9].

The general idea is to apply a chain of linear transformations to the x_i vector, each followed by the application of a non-linear *activation function*, which ensures that each layer is not a linear transformation itself.

A first surprising result about neural networks is that they are a *universal function approximator*, which roughly means that most functions can be reasonably well approximated by a neural network. This is particularly important for supervised learning, which deals with unknown relationships between inputs and outputs. Indeed, if there is a reasonable function between the data and the labels ($y_i \approx f(x_i)$), then one can find weights for a neural network that makes it behave as the function f.

A second result about neural networks is that several of them can be averaged weight-wise to obtain a new network that performs approximately as well as if it were trained on all datasets involved. This enables parallelization of ML tasks on different physical nodes, each with its own dataset: each node trains a local model, shares it with the others and averages all of the received models into one aggregated model. Ignoring communication concerns, this yields a learning time speedup that is linear in the number of nodes in the network. There are several kinds of architectures of such distributed learning systems, which we detail in the next section.

2.2 Distributed learning systems

Distributed machine learning is a set of techniques that diverges from centralized methods by distributing the responsibility of training across multiple computing nodes. In contrast to traditional approaches that aggregate all data for training in a single location, distributed machine learning allocates data and computation among various nodes. This decentralized framework enhances scalability, efficiency, and privacy, making it suitable for applications dealing with extensive datasets and prioritizing data locality.

The subsequent paragraphs delve into two specific aspects of distributed machine learning: federated learning and decentralized learning. They differ in the way individual workers share their updates with the others.

2.2.1 Federated learning

Federated learning (FL) is one of the main paradigms of distributed ML. It makes use of a central server that is tasked with the aggregation and distribution of models from the worker nodes, but does not perform training work *per se*.

In a typical FL training round, workers obtain the model from the central server, update the model by a fixed amount of training on their local dataset and share the updated model back to the server. The server itself is usually assumed to be perfectly available, as nodes have no way of communicating with each other otherwise. While this requirement of availability is fairly strong, it enables a very fast convergence of model updates: the training done by some worker will be included in the next round of every other worker, meaning that worker nodes will have less of a tendency to overfit on local data.

More precisely, we will assume in this report that FL behaves as described in the following algorithms:

Algorithm 2.2.1 Server-side Federated Learning

```
current_model ← random initialization;
for iteration ∈ {0..max_iters} do:
    for node ∈ workers do:
        send current_model to node;
    end for
    for node ∈ workers do:
        received[node] ← await answer from node;
    end for
    current_model ← aggregate(received);
end for
```

Algorithm 2.2.2 Worker-side Federated Learning

```
for iteration ∈ {0..max_iters} do:
    current_model ← await server model;
    current_model ← train(current_model, dataset[iteration]);
    send current_model to server;
end for
```

While the server-side aggregation method is left unspecified, it is typically done by averaging weights in the case of neural networks, which is how we will use FL in this report.

2.2.2 Decentralized learning

The main alternative to FL is called **Decentralized Learning (DL)**. It removes the assumption that a centralizing server be available at all times. Instead, each individual node has the responsibility of both training its model and propagating its updates to its assigned group of other neighboring nodes.

The main idea behind this configuration is that each node essentially behaves like a 'server' for its neighbors. This has the benefit of having no critical dependency on a single machine, as a failure of a given node is mostly quarantined to its neighborhood set. However, model updates tend to propagate fairly slowly, which means that a model in a given worker node tends to perform better on data located on a node that is close in the neighborhood graph. Conversely, local models might poorly generalize to data that is located on a far worker node. This means that DL systems might display a tendency to overfit on their local training data, more so than their FL counterparts.

In this report, we will use the following DL algorithm:

Algorithm 2.2.3 Decentralized Learning

```
current_model ← random initialization;
for iteration ∈ {0..max_iters} do:
    for node ∈ neighbors do:
        send current_model to node;
    end for
    for node ∈ neighbors do:
        received[node] ← await answer from node;
    end for
    current_model ← aggregate(received);
    current_model ← train(current_model, dataset[iteration]);
end for
```

Chapter 3

Experiment design

As mentioned in the introduction, we strive to study how well models generated by FL and DL behave when they are swapped into the other algorithm for further training. We therefore design experiments that reveal the influence of the various parameters at play.

3.1 Initial performance comparison

The first (and main) experiment consists of training an ML model using federated or decentralized learning for a fixed amount of iterations, and reusing that partially trained model in the other algorithm for the rest of the experiment, effectively 'swapping' once (and once only). This simulates a central server definitive crash or recovery, where the section during which the server is unavailable is run in DL and the other in FL.

There are therefore two parameters to vary: the iteration at which we swap the algorithms (which we call the **swap point** of the experiment), as well as the graph structure used by the DL algorithm when it is actively running. We also measure the performance of pure FL and DL as a control experiment, which are equivalent to a swap after the last iteration or before the first.

3.1.1 Decentralized graph

It is reasonable to expect the performance of decentralized learning to be influenced by the density of the underlying graph of neighbors. We therefore run experiments on three different graphs:

- A ring (i.e. connected 2-regular) graph, which is the least dense among all connected regular graphs.

- A $\lceil \ln |V| \rceil$-regular graph, which is reasonably dense.
- A $|V|/2$-regular graph, as a demonstration of an extremely dense graph.

Note that we experiment with an $|V|/2$-regular graph and not a fully connected graph, as DL and FL are indistinguishable in that case.

This variation in the graph density is meant to display a gradual transition, from DL runs that are close to federated (highly dense graphs) to runs that are further away from FL behavior (sparser graphs). While it is empirically clear that denser DL network architectures perform better than sparser ones, it is possible that sparse network architectures can exploit pre-trained FL models about as well as their dense counterparts. If that is the case, we should prefer them because of their lower communication costs. This justifies why we investigate the importance of graph density when swapping models.

3.1.2 Server crashes and recoveries

Orthogonally to the graph density, we also study the influence of the position of the swap point. For example, we can place the swap point very early, effectively giving the second algorithm more running time at the expense of the first.

The swap point variations will show how much the current algorithm matters to the final performance of the model, relatively to the past training. Assume, for example, that the experiment at play is running DL first, followed by FL after a swap point at iteration n. If at iterations occurring right after n the model performed closely to DL, we could conclude that the performance of the model is mostly explained by its past training. If however we observed that the model performance rejoins that of 'pure' FL, we would conclude that the algorithm that is running after the swap dictates performance by itself, and the algorithm running before the swap is essentially not relevant. As we will see, in practice, the degree to which the second algorithm influences the final model performance relatively to the first one depends non-trivially on the position of the swap point.

When running our experiments, out of the 2000 iterations we run, we place a swap point at one of iterations 100, 750, and 1500. When an experiment starts with FL and is followed by DL, we call this experiment a **crash experiment**, in reference to its correspondence with a hypothetical server failure that caused FL to become inoperable. Conversely, when we swap from DL to FL, we call the experiment a **recovery experiment**.

3.1.3 The experiment with its parameters

Once we have fixed a graph, a swap point, and one of the two orderings of FL and DL in the experiment, we will run the algorithms as described above for a total of 2000 training iterations, which are sufficient for decent convergence of both the original versions of FL and DL. More details about the training dataset and hyper-parameters used are provided in the Implementation section. We will measure the testing loss and accuracy every 50 iterations. Accuracy in particular will be our main performance ranking metric.

3.2 Deeper analysis around server crashes and recoveries

As we will see in the Results section, the experiments described above show unexpected behavior around the swap points. We therefore investigate the performance of the model around the swap points in more detail, measuring the testing losses and accuracies at every iteration for 50 iterations after a swap. This will give more insight on the influence of swaps on models, which was not covered by the previous experiment.

Chapter 4

Implementation

In this section, we provide the implementation details that are necessary for the reproduction of the results obtained in the report. It presents the tooling we used for the experiments, a proof-of-concept algorithm that performs the training with automated DL-FL swaps, as well as the hyperparameters used in the training of node-local ML models.

4.1 DecentralizePy

We use DecentralizePy[3], a Python framework for distributed learning, developed at EPFL. DecentralizePy provides functionality for the development and evaluation of distributed ML algorithms. It notably includes modules for node-to-node communication, node-side ML algorithms, and a network stack for peer-to-peer and client-server communication. We specifically leverage the network stack to implement our experiments and use this implementation to evaluate the different cases.

4.2 Unified algorithm

While DecentralizePy already provides working implementations of both FL and DL, they are not able to exchange models across algorithms by default. This is why we introduce a hybrid algorithm that may perform swaps at will, in order to perform self-contained experiments. For the sake of conciseness, we will nickname this algorithm **Swap Learning** for the rest of this report.

4.2.1 Pseudocode and relationship to FL and DL

First, let us give a first draft of the mentioned algorithm, which first attempts to communicate with a central server, and uses DL with a predetermined list of neighbors as a backup in the case of a server failure. Naturally, since we are concerned with improving FL while embracing server unavailability, the server part of the algorithm remains the same, and the worker-side algorithm is adapted to handle the case where the server is unavailable. Algorithm 3.1.1 summarizes this definition.

However, we quickly noticed that this approach suffers from a flaw that exists in neither FL nor DL: when the server recovers from a crash, its model is old yet has priority over the newer models trained in the meantime. Under these conditions, all the work performed while the server was down would be wasted. We thus introduce a worker-side flag that signals whether the last iteration was successful in contacting the server or not. When receiving a model from the server, workers check whether their last iteration also used a server model. If not, the local model is used instead of the server one for a single iteration, so that the server may update its own model with up-to-date weights. Algorithm 3.1.2 summarizes the patched version.

It is important to notice two properties: Swap Learning behaves exactly like FL when the server is always available, and exactly like DL when the server is always down. This will be used extensively in the experiments we run.

Algorithm 4.2.1 Worker-side Swap Learning (naive approach)

```
current_model ← random initialization;
for iteration ∈ {0..max_iters} do:
    if server is reachable then :
        current_model ← server model
        current_model ← train(current_model, dataset[iteration]);
        send current_model to server;
    else:
        for node ∈ neighbors do:
            send current_model to node;
        end for
        for node ∈ neighbors do:
            received[node] ← await answer from node;
        end for
        current_model ← aggregate(received);
        current_model ← train(current_model, dataset[iteration]);
        send current_model to server;
    end if
end for
```

Algorithm 4.2.2 Worker-side Swap Learning (patched)

```
current_model ← random initialization;
last_iteration_success ← True;
for iteration ∈ {0..max_iters} do:
    if server is reachable then :
        if last_iteration_success then
            current_model ← server_model;
        end if
        current_model ← train(current_model, dataset[iteration]);
        send current_model to server;
        last_iteration_success ← True;
    else:
        for node ∈ neighbors do:
            send current_model to node;
        end for
        for node ∈ neighbors do:
            received[node] ← await answer from node;
        end for
        current_model ← aggregate(received);
        current_model ← train(current_model, dataset[iteration]);
        send current_model to server;
        last_iteration_success ← False;
    end if
end for
```

4.2.2 The oracle

Assuming that we base the server unavailability on timeout-based methods, this approach falls short when considering determinism across experiment environments. It suffers from the sometimes whimsical schedulers and arbitrary physical network slowdowns which may severely influence the quality of learning by making worker nodes erroneously time out, when the server model was in network transit. To account for this, we also introduce a global oracle in the testing code that determines whether the server should be reachable. If the oracle predicts a server unavailability, the worker will not wait for the server model. If it does not, the worker sets an arbitrary large (or infinite) amount of time as a timeout, ensuring that models sent by the server will be correctly received. The global oracle is adjusted for every experiment so that it may simulate server crashes at precise points in time. This oracle always results in an interleaving of events that is possible without its presence, and helps to maintain strong guarantees about experiment determinism.

4.2.3 The final implementation of Swap Learning

The code we use for the experiment is hosted here.[1] In particular, the implementation of the worker-side Swap Learning algorithm is located here.[2]

4.3 Experiment details and hyperparameters

For the sake of reproducibility, we ran all experiments with as many shared parameters as possible. First of all, the amount of worker nodes was systematically 35, not including the server.
All workers were given a non-iid fraction of CIFAR-10 as their training data. The data followed the 4-shards distribution for non-iid-ness.
The model used for training is a LeNet neural network implemented in PyTorch[8] as described here.[3] The random seed provided to DecentralizePy for experiment determinism was systematically set to a value that is constant across experiments.
Most if not all other hyperparameters that might be of interest to the reader are listed here.[4]

[1]https://gitlab.epfl.ch/randl/decentralizepy/-/tree/masters-mathis?ref_type=heads
[2]https://gitlab.epfl.ch/randl/decentralizepy/-/blob/masters-mathis/src/decentralizepy/node/DPSGDNodeTimeout.py?ref_type=heads
[3]https://gitlab.epfl.ch/randl/decentralizepy/-/blob/masters-mathis/src/decentralizepy/datasets/CIFAR10.py?ref_type=heads#L280
[4]https://gitlab.epfl.ch/randl/decentralizepy/-/blob/masters-mathis/tutorial/config_celeba_sharing.ini?ref_type=heads

Chapter 5

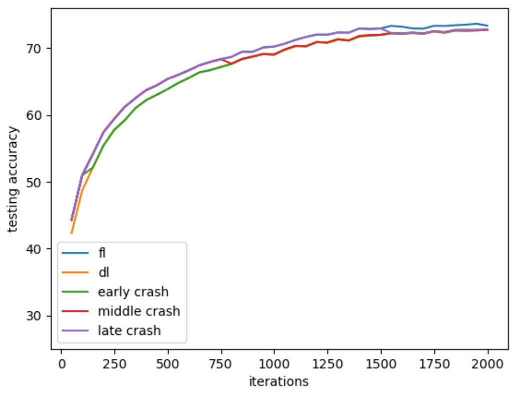
Results

5.1 The main experiment

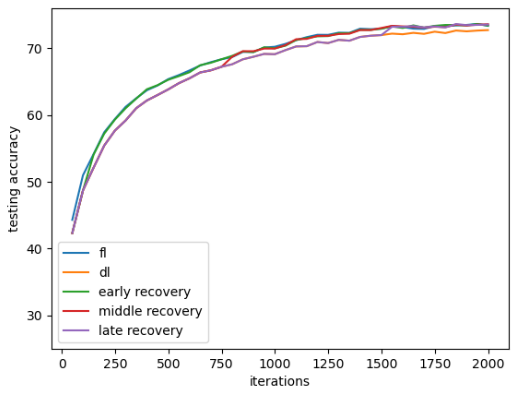
Figures 5.1 and 5.2 sum up the accuracies and losses measured in the main experiment.

The most visible result, and by far the most surprising, is that no matter the graph structure, the swap point, or the starting algorithm, *both accuracy and loss are almost exclusively determined by the algorithm that is running at the time of their measurement.* In the case of 18-regular and 4-regular graphs, it is virtually impossible to tell the end of runs apart from each other. This is somewhat less true in the ring graph, but they remain impressively close.
For recoveries, this result is essentially a testimony to how good of an idea it is to average neural networks weight-wise, in order to obtain a new neural network that performs as if it were trained on all datasets involved: individual nodes are performing well on their own data and that of their close neighbors before the recovery point. After that, the switch to FL-like behavior averages all models into one that immediately rejoins the performance of the control FL model. Since the testing set is made of classes which all belong to the distribution of at least one worker node, the aggregated model is able to correctly classify samples from all classes.
For crashes, it appears that the opposite holds: individual nodes overfit on their data as soon as they are given the occasion. This results in the sizeable drop in accuracy (and rise in loss) seen in the ring architecture, which is the furthest to FL. Since the performance obtained over time has diminishing returns, this means that a swap from FL to DL causes long-lasting damage to both accuracy and loss that is overcome with great difficulty, if it even ever happens.

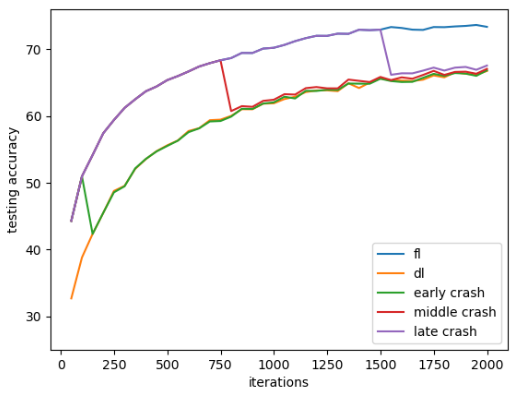
We also note that *it does no harm to run more iterations in FL.* It is always the case here that more training in the FL-side of the algorithm leads to better performance: the plots occasionally overlap but never intersect. It even sometimes helps quite significantly, as seen in the ring architecture, where longer training in FL yields visibly better results.



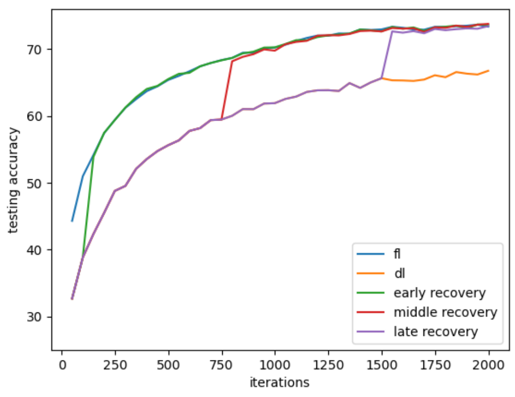
(a) 18-regular crashes



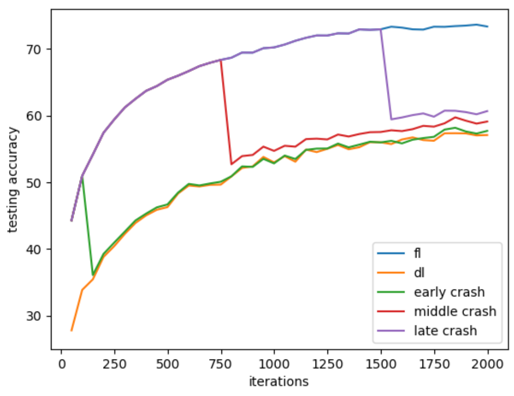
(b) 18-regular recoveries



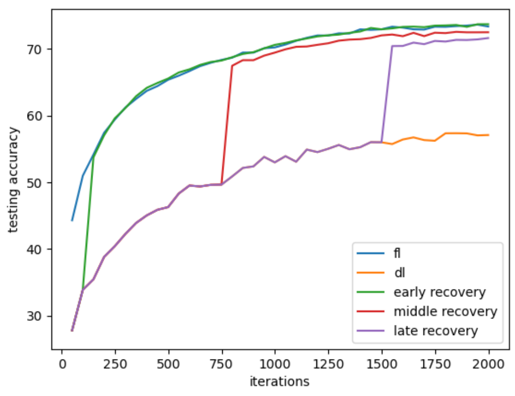
(c) 4-regular crashes



(d) 4-regular recoveries

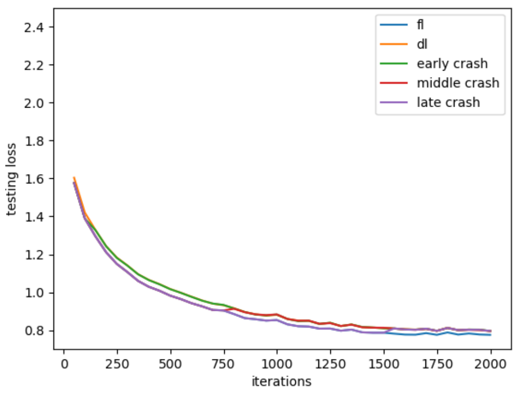


(e) Ring crashes

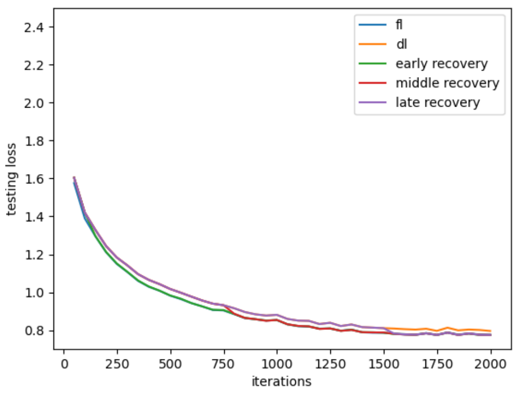


(f) Ring recoveries

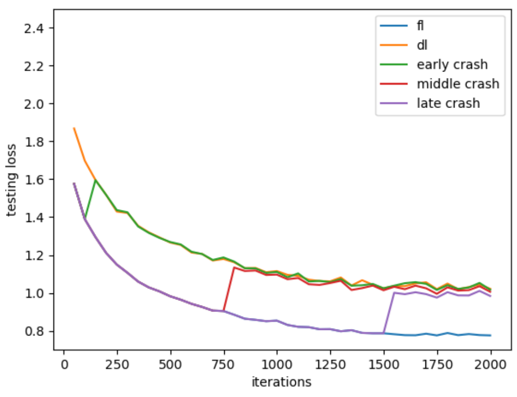
Figure 5.1: Accuracies w.r.t. iteration number



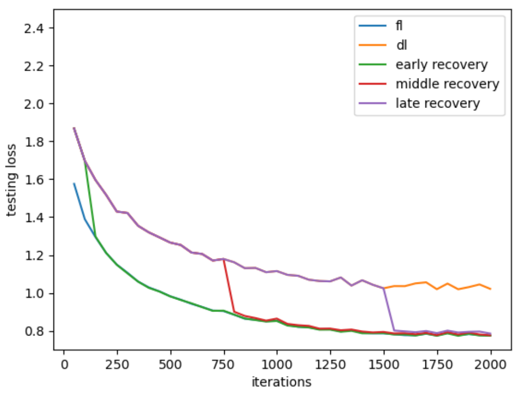
(a) 18-regular crashes



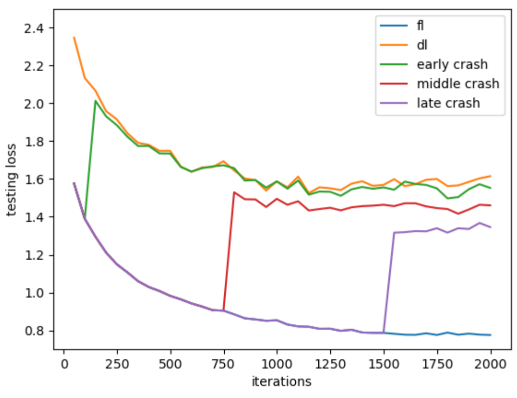
(b) 18-regular recoveries



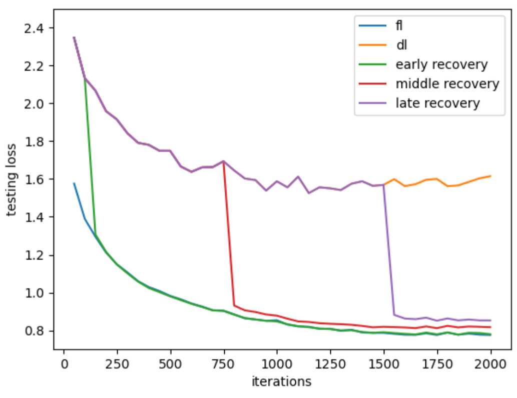
(c) 4-regular crashes



(d) 4-regular recoveries



(e) Ring crashes



(f) Ring recoveries

Figure 5.2: Losses w.r.t. iteration number

5.2 Zooming in on the transition period

Figure 5.3 displays the accuracies obtained when running the exact same experiment, but sampling more test performance data around the swap point at iteration 750. It is clear that both crashes and recoveries display very similar behavior across network configurations.

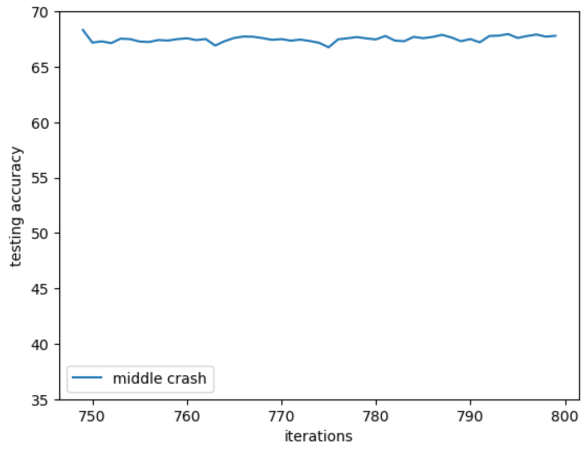
For crashes, the drop in performance is essentially immediate. This was unexpected, we thought the performance drop would be more gradual in time. This suggests the interesting fact that over-fitting on a worker node occurs instantly as soon as the occasion is given. This is clearly a function of the hyperparameters at play: if nodes trained on way fewer samples before communicating, they could not have such a strong influence of the model. However, we think that the hyperparameters used have very reasonable default values, which are extensively used at large. This effect is then representative of real systems.

Recoveries are also immediately affected by the swap. The interesting spike downwards seen at iteration 750 is essentially an implementation artifact mentioned in 4.2.1: the server has too old a model after a recovery, so nodes ignore the first model they receive and train on their own model instead. This is yet another suggestion that models will overfit if they are given the chance. During iteration 751 and afterwards, the implementation behaves like FL, as intended. It is then very clear that the benefits of swapping to FL are immediate. This is a surprisingly efficient method for aggregating the independent training of worker nodes, even if said training involves very large amounts of data since the last global model synchronization.

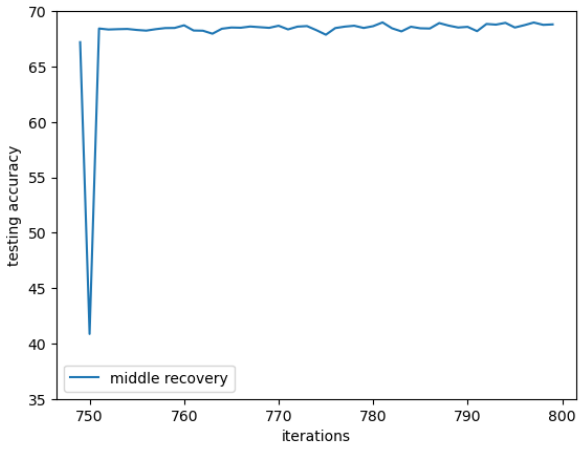
5.3 Improvement on DL: do the last iteration in FL

Since we just established that recoveries enable *immediate and massive* performance boosts, we suggest to exploit this by running a federated learning instance and averaging all weights at the end of the algorithm before the final performance evaluation. Results of this experiment are displayed in Figure 5.4.

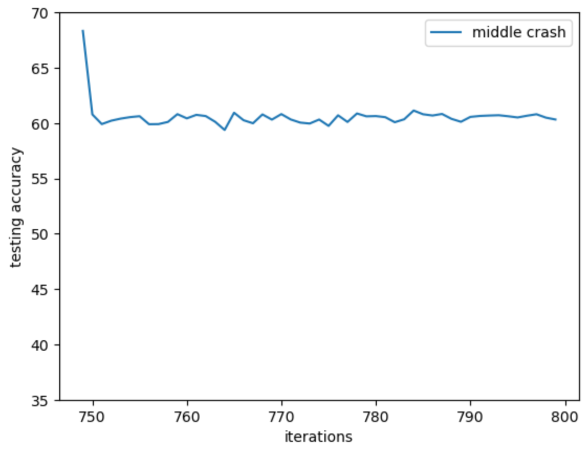
The results are essentially the same as those of section 5.1, but pushed to the limit of the iteration numbers. The effect is particularly pronounced for sparser graphs, which usually have a very poor DL performance when compared to FL. We can then effectively bridge the gap between both algorithms using a simple global average. While it may be expensive in DL-friendly setups to have a global server able to handle the single averaging of a large amount of models, we note that this process can be distributed by having nodes repeatedly average their weights with that of their neighbors without training. Using a symmetry argument, it is clear that this process will converge at the same point as what FL would have produced. This suggests that most real-world uses of DL are likely to heavily benefit from this process.



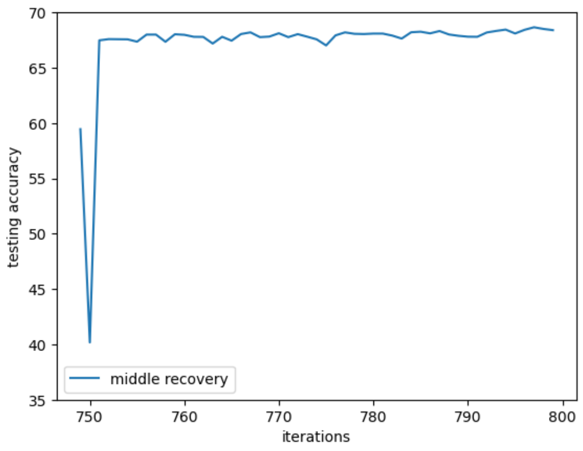
(a) Zoom on 18-regular crash



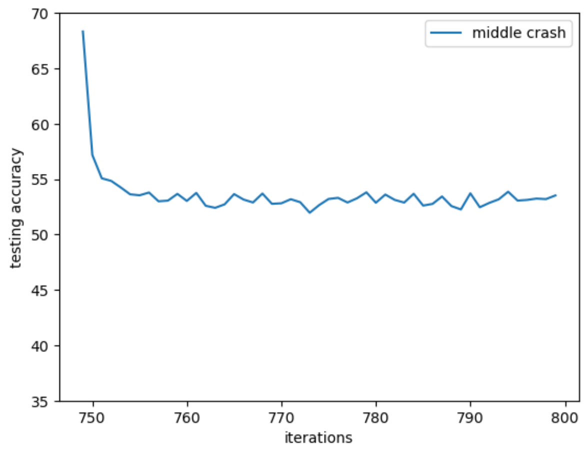
(b) Zoom on 18-regular recovery



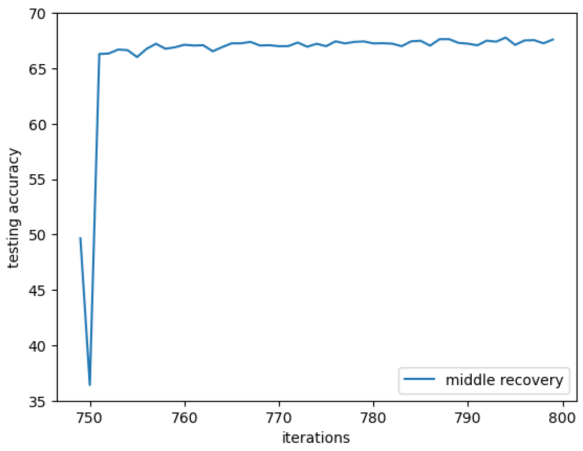
(c) Zoom on 4-regular crash



(d) Zoom on 4-regular recovery

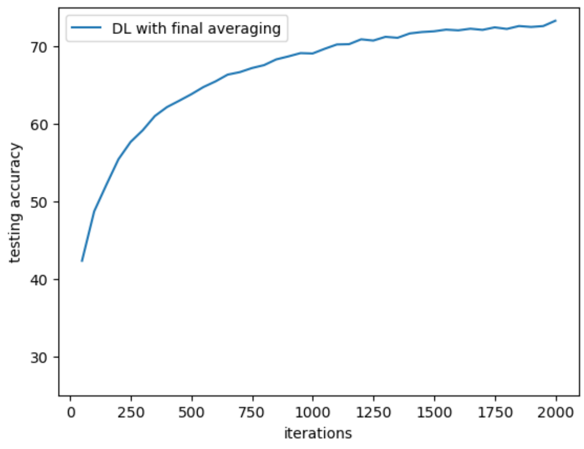


(e) Zoom on ring crash

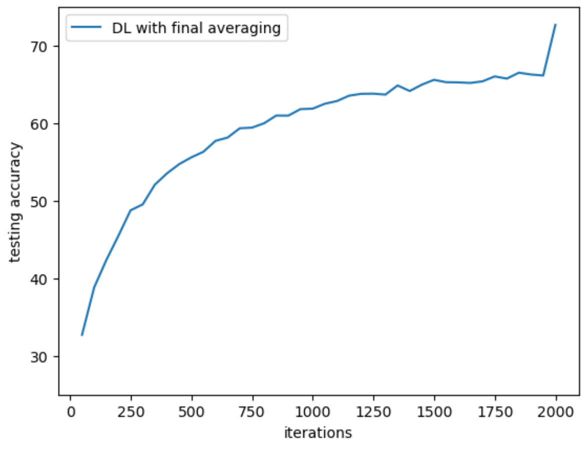


(f) Zoom on ring recovery

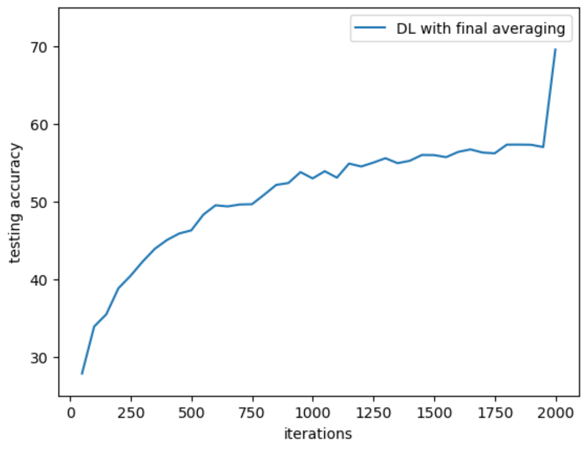
Figure 5.3: Testing accuracies of crashes and recoveries around the swap point



(a) 18-regular graph



(b) 4-regular graph



(c) Ring graph

Figure 5.4: Testing accuracies of DL 'upgraded' with a final round of averaging

5.4 Reproducibility statement

All experiment logs used for the production of the plots above are available at the request of the reader. The reader may obtain the same logs by running the code mentioned in the Implementation section.

We observed that while experiments replicate deterministically on a given machine, they do not match exactly across different machines, but are always extremely close. We blame this on floating-point approximations and hardware/scheduler implementations.

We therefore expect that the reader should be able to re-create the same results as the ones presented here.

Chapter 6

Conclusion

The results above give us enough information to propose the following answers to our research questions.

6.1 Is Swap Learning viable on its own?

Swap Learning was originally meant as a unified environment for studying the ’swap points’ experiments. It appears that using it as a standalone algorithm does not immediately make sense. In the case of a server failure, Swap Learning switches to decentralized mode, and its performance drops to match that of DL. That is usually much lower than FL (depending on graph density), and so simply stopping the training immediately after the crash would have resulted in a better model obtained in a smaller amount of iterations. We therefore cannot recommend the deployment of Swap Learning on real-world machine learning applications in the case of single swap points.

However, we did not study the case of multiple swap points, which may prove interesting for cases like a server crash followed by a recovery. Given the observed behavior on single swap point experiments, one might reasonably expect FL-like performance to be reachable even though a large number of iterations would not be done in federated mode. This will be the object of future research.

6.2 The relationship between DL and FL

This report suggests that DL and FL are more closely related than initially expected: in most scenarios, the high performance of FL is reachable for DL models even after a small number of centralized averaging rounds. This holds (1) reliably across very different architectures and (2) immediately after the swap between the two algorithms. Since FL can reasonably faithfully be emulated by DL and a few rounds of averaging near the end, as underlined by this report, we propose the following improvement to DL:

6.3 DL on steroids

After a DL training run, averaging once the models obtained so that all nodes share the same model improves the performance dramatically, at a very low cost. In some scenarios, it even matches FL without the requirement for a central server to be available at all times. Since the final averaging can equivalently be done in a decentralized manner, a server is in fact not needed at all.

Bibliography

[1] Horace B Barlow. “Unsupervised learning”. In: *Neural computation* 1.3 (1989), pp. 295–311.

[2] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. “Supervised learning”. In: *Machine learning techniques for multimedia: case studies on organization and retrieval.* Springer, 2008, pp. 21–49.

[3] Akash Dhasade, Anne-Marie Kermarrec, Rafael Pires, Rishi Sharma, and Milos Vujasinovic. “Decentralized learning made easy with DecentralizePy”. In: *Proceedings of the 3rd Workshop on Machine Learning and Systems.* 2023, pp. 34–41.

[4] István Hegedűs, Gábor Danner, and Márk Jelasity. “Decentralized learning works: An empirical comparison of gossip learning and federated learning”. In: *Journal of Parallel and Distributed Computing* 148 (2021), pp. 109–124.

[5] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[6] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. “A review of applications in federated learning”. In: *Computers & Industrial Engineering* 149 (2020), p. 106854.

[7] Batta Mahesh. “Machine learning algorithms-a review”. In: *International Journal of Science and Research (IJSR).[Internet]* 9.1 (2020), pp. 381–386.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[9] Sun-Chong Wang and Sun-Chong Wang. “Artificial neural network”. In: *Interdisciplinary computing in java programming* (2003), pp. 81–100.