A Performance Evaluation of a Hybrid Approach to Collaborative Learning

Mathis Randl

June, 2022

Abstract

Machine learning based on distributed systems has gained traction as a way to help solve the problems of privacy, data transfers and computing power of traditional centralized methods. Two main methods have emerged out of this approach: federated and decentralized learning. In this report, we introduce a timeout-based approach to collaborative learning systems that lies between these two methods. We evaluate the influence of several network parameters over its convergence time, and then compare its performance against that of the other two in various network conditions. We discover that the algorithm is particularly sensitive to the packet loss rate of the network. It also outperforms both alternatives in terms of time taken by iteration when the network is in a particularly difficult configuration. However, its testing accuracy grows at a slower pace per iteration than its counterparts, leading to a performance trade-off when designing collaborative systems in difficult network configurations.

1. Introduction

Recent years have seen a rapid increase of the collection of massive amounts of data, through the popularization of IoT devices and end-user data collection in general. This, in turn, proved to be a challenge for centralized machine learning pipelines and recommendation systems, for several reasons: data privacy concerns legally prevent the unrestricted sharing of user private data, and the growth in available computing power in a single system did not manage to follow that of the size of the available datasets. This led to the promotion of alternative systems based on **collaborative learning**, which make use of distributed systems to avoid sharing data across the network. In this configuration, nodes that want to participate in model training need not share their own dataset with the rest of the system. This solves both of the main issues of data centralization. These collaborative learning systems are organized in two main types, depending on whether they make use of a central server to collect and distribute weights.

While one (**federated learning**) expects that a central server be available to aggregate the progress of worker nodes, the other (**decentralized learning**) does not, at the cost of putting the responsibility of propagation to the worker nodes. More details about these two classes of algorithms can be found in the Background section.

In this paper, we propose an alternative to these two classes by designing a hybrid algorithm that attempts to share progress with a federating server whenever the underlying network conditions allow for it, but falls back to a decentralized mode of functioning when the conditions are too harsh. We expect this to bring several benefits: whenever possible, this algorithm profits from the fast communication of work to all nodes through the main server, but still offers a possibility of learning in a collaborative manner when that server becomes unavailable. The problem of overwhelming either the network interface or computing power of the centralized server is then entirely eliminated.

1.1. Objectives of research

The research we will then conduct is the following: First, we will **design an algorithm**

that combines the strengths of both decentralized and federated learning using timeouts to make a decision about which technique to use (we will call that algorithm Timeout). Then, we will **measure the influence of several network parameters on this algorithm**. Finally, we will **benchmark the algorithm against standard implementations of decentralized and federated learning** to understand its strengths and weaknesses when compared to more standard collaborative learning algorithms.

2. Background

2.1. Federated learning

The first algorithm we will present here is Federated learning. The federated worker algorithm consists of three main steps: first, it awaits receiving parameters from server. Then, it locally trains on received parameters, and finally send updated weights to server. It repeats this for a set number of iterations. The server algorithm is the dual of the worker one: it handles the distribution of weights and receives the updated weights from the workers. When it has received messages from all workers, it aggregates the received weights into one model, typically by averaging neural network weights and biases.

There are several variations to this algorithm: averaging is only one of the available ways to reduce several models into one, and workers can send the gradient instead of the updated weights, which leaves the weight update to be done by the federating server. In this report, we will work with the version we initially presented, as it is the one already implemented in the Decentralizepy framework. An important fact about this implementation is that none of the server and worker nodes ever timeout on their respective waiting steps. This means that the classification or regression performance of the final model is independent of network quality, but poor network conditions may have a large negative impact on the amount of training iterations accomplished by unit of time.

In general, this approach suffers from the centrality of the server, notably if the federating server fails or becomes unreachable for a part of the nodes. This is a problem that the second algorithm addresses.

2.2. Decentralized learning

The second algorithm of interest here is Decentralized learning. It solved the problem of having a single point of failure by scrapping the asymmetry in the roles played by the nodes, opting for a peer-to-peer approach where all of them have access to their own dataset, and are responsible for both model training and weight sharing with their peers. The algorithm follows three steps: first, the nodes train on their private data and send the updated model to their neighbors. Then, they receive the model from their neighbor and aggregate it with theirs. They repeat this for a fixed number of iterations.

By having the nodes only share their weights with a set of neighbors (which may be static or dynamic), the nodes are not overloaded in their exchange, avoiding the main pitfall of federated learning. However, it may suffer from the lack of communication between nodes that are not close neighbors, as it will take several iteration steps for the training information of a node to reach another one that's far away in terms of neighbor hop distance. Conversely, federated learning guarantees that training progress is transmitted to every node at every iteration.

3. Method

3.1. Tooling

The performance evaluation of the algorithm we propose will make use of two main tools: Decentralizepy and Kollaps.

3.1.1. Decentralizepy

Decentralizepy[1] is a Python framework designed to run decentralized machine learning applications. It was built to analyze several aspects of collaborative learning, from measuring the time it takes to run a decentralized learning round to evaluating the performance of a model that was trained in a decentralized fashion.

It is written in a way that abstracts away several key concepts of decentralized learning. For example, it automatically manages the communication setups with peers, the network communication protocols and the machine learning training. The part of this high-level framework that is of most interest to this report is its implementation of the decentralized and federated learning algorithms. We used these as templates to create our timeout-based algorithm that is inspired by both.

In order to understand how the network quality influences how the algorithms manage to learn, however, the framework alone is not enough: it offers little control over the underlying network. We cannot, for example, control the maximum I/O throughput from within the application. This is why we introduce another tool that enables fine-tuning of these low-level parameters independently of Decentralizepy.

3.1.2. Kollaps

Kollaps[2] is a network simulation tool that can run any distributed application on a single computer or computing cluster. Every instance of the application runs in its own Docker container, and may run arbitrary code from an arbitrary language. The containers run at the maximal speed allowed by the underlying hardware, which Kollaps does not control, but it then enables fine-tuning the quality of the connection that join containers together.

To describe experiments to Kollaps, one must provide a pre-compiled topology file containing a list of containers, along with the command to run and its dependencies in the form of a Docker image. This topology file also includes a description of the network to emulate: switches, links and timed events like artificial crashes. When declared, links are given a certain capacity, throughput and delay that are the main tools to simulate physical network limitations.

In this project, Kollaps was used in the following way: several containers were spawned to run Decentralizepy-based applications, and each ran its own instance of the framework, using the others as peers. By modifying the network quality joining the containers, we could

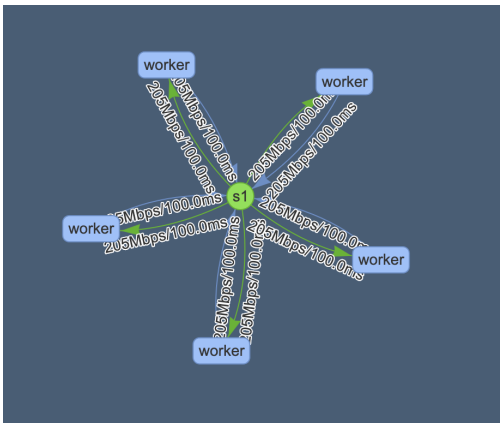

Figure 1: Screenshot from the Kollaps dashboard, showing a star network with five members linked by a switch. One of the workers is a federating server. Link characteristics (max. throughput / packet delay) are shown on the links.

then extract meaningful information about how network quality influences the convergence time of the algorithms. More specifically, we varied the maximum throughput, delay and packet loss of all the links joining the containers, while benchmarking several distributed algorithms to gain insight into their differences and similarities.

For most experiments, the main setup was the same: we used four worker containers in a star-network centered around a switch. Occasionally, a fifth container was needed to serve the role of a federating server if one was required by the algorithm. We then placed a link between each container and the switch, as described in Figure 1.

3.2. Benchmarks

We ran three benchmarks on Decentralizepy using Kollaps. The first one aims at understanding which network characteristic is the most harmful to performance in the timeout-based algorithm we introduce in this report. The second one compares the convergence speed of the three algorithms (decentralized, federated, timeout-based) in the wild over several network configurations. The third one is a sidenote on the accuracy of the classifiers produced in these experiments.

3.2.1. First benchmark: behavior of Timeout

We first tried to isolate the effect of three network characteristics on the Timeout algorithm: bandwidth, delay and drop rate. To achieve this, we ran the exact same Decentralizepy experiment on twenty-seven different network setups, where we varied between all combinations of what is considered a low, mid and high value for a given network characteristic. Here, the algorithm ran for 80 training iterations, which proved sufficient to reliably observe differences in running time. Table 1 shows the values used in this benchmark.

Characteristic	Best	Avg.	Worst
Throughput (Mbps)	400	210	20
Delay (ms)	4	200	400
Drop rate	0%	2.5%	5%

Table 1: Network characteristics used for the first benchmark

These values mostly come from the Diablo[3] paper, which also benchmarks decentralized applications using connection speeds and delays between computers located around the world. In particular, a value of 20 and 400Mbps both are a rounded approximation of the lowest and highest connection throughput found in the pair of cities listed in the Diablo paper. The same goes for the worst delay, which is the worst possible in the list. The best delay of 4ms is due to simulation limits: Kollaps does not simulate sub-millisecond ping, and we need four hops to go from one node to the other and back through a switch. Finally, the worst value for the drop rate was chosen arbitrarily, as we found no academic source on typical packet drop rates over the Internet. However, we claim that a 5% drop rate is largely sufficient to understand the effect of drop rate on the experiments we run. For all characteristics, we also included the average value between the two extremes, which provides a good idea of whether the influence of a network characteristic over the running time is linear.

Name	Throughput	Delay
Good-net	400Mbps	2ms
Mid-net	50Mbps	150ms
Bad-net	10Mbps	250ms
World-net	Mixed	Mixed

Table 2: Network characteristics used for the second benchmark

3.2.2. Second benchmark: speed of convergence comparison

We then wished to understand how fast these three algorithms converged relative to each other. For this, we designed a series of experiments on a list of different networks, varying again the three components of network quality. We then ran the three algorithms on these varying networks and measured the time of convergence and iterations per unit of time. Again, we deemed 80 training iterations to be sufficient.

The network parameters are similar to that of the first benchmarks, with some modifications. Since this benchmark runs several algorithms, we reduced the number of experiments per algorithms to avoid a combinatorial explosion of the amount of experiments to run. This is why the throughput and delay are merged together in the notion of network setup (good-, mid-, and bad-net). Table 2 summarizes the different qualities of networks used in the second benchmark.

This benchmark however introduces a new concept, that of *world-net*. This is a network simulation with asymmetrical links: nodes are connected by links whose quality is meant to simulate computers located around the world, connected through the Internet. This implies that link quality is not the same across pairs of nodes, and we picked them to be as geographically apart as possible. The cities picked for the simulation are Milan (Italy), São Paulo (Brazil), Cape Town (South Africa), and Tokyo (Japan). When a federating server is required, it is simulated as being located in Stockholm (Sweden). The source for the quality of connection between pairs of cities is again the Diablo paper, which specifies the values copied in Table 3.

Cities	Throughput	Delay
Stockholm - Milan	404Mbps	30ms
Stockholm - Tokyo	42Mbps	241ms
Stockholm - C. Town	60Mbps	179ms
Stockholm - São Paulo	48Mbps	214ms
Milan - Tokyo	48Mbps	214ms
Milan - Cape Town	67Mbps	162ms
Milan - São Paulo	49Mbps	211ms
Tokyo - Cape Town	26Mbps	354ms
Tokyo - São Paulo	39Mbps	256ms
C. Town - São Paulo	27Mbps	340ms

Table 3: World-net characteristics, from the Diablo paper

3.2.3. Sidenote: performance in terms of testing accuracy

Since training is done in a synchronous manner in both traditional algorithms, network connection quality is of little importance for the final performance of the trained classifier. That's why this metric is of small importance to this report, which studies the influence of network parameters over the running time of algorithms. Nonetheless, we also included a small benchmark of the testing accuracy of the model per iteration produced. This will enable studying the degree to which algorithm iterations improve the training accuracy.

For this, we ran all three algorithms on good-net with high packet drops for 200 iterations (as opposed to the previous 80), while measuring their testing accuracy at every iteration. Since measuring the performance over the testing set at every iteration is fairly time-consuming, we will only run one such experiment.

4. Obtained Design and Results

4.1. Algorithm design

The Timeout algorithm which we propose has two parts: the aggregating server (in the same line as federated learning), along with worker nodes that first attempt to contact the federating server, in order to obtain the model of the current iteration. Depending on whether they succeed in getting the model, they share it with their neighbors or attempt to retrieve the model from their neighbors. Finally, they train on it and attempt to send the model back to the federating server. The Timeout server essentially does the same thing as the server in federated learning: it sends its own model, waits for the worker nodes to update it, and aggregates the updated models. It has a small difference from its federated counterpart: if it waits for a message for too long, it will time out and skip to the next iteration, so that it can follow fast-working nodes if one of the other workers is too slow.

The pseudocode for the worker nodes we programmed goes as follows:

```
repeat for a fixed amount of iterations:
  - attempt model retrieval from server
  - if retrieved (without timeout):
  |   set local model to server model
  |   broadcast model to neighbors
  - else:
  |  broadcast model to neighbors
  |  await neigbor models with timeout
  |  aggregate local model with answers
  - train on the new local model
  - send to server
```

and the server runs the following code:

```
initialize model at random
repeat for a fixed amount of iterations:
  - send model to every worker
  - await their answer for a certain time
  - aggregate all received answers
    with the current model
```

The full Python implementation is available on GitLab[1].

The point of this algorithm is to be an extension to federated learning that handles poor network connectivity properly. Indeed, if the network behaves ideally, the model retrieval will systematically succeed, as well as the last step where the worker node sends back the updated model to the server. This means that the steps executed will be the exact same as those executed by federated learning. However, it has a defined fallback behavior when

[1] https://gitlab.epfl.ch/randl/decentralizepy/-/tree/timeout

the network behaves improperly: it starts decentralizing the learning process.

Since it re-evaluates the network connectivity at every iteration, it can also handle variation in network conditions. For example, networks with high jitter (Wi-Fi[4], ...) will create cases where some iterations are able to connect to the federating server, and others not. In this case, the algorithm attempts to remain as close as possible to federated learning, and falls back to decentralized learning only when reaching the server is not possible at this iteration.

A consequence of this fact is that different nodes might be running different kinds of iterations at the same time: some nodes may have managed to connect to the server, while others could not. This is why even in case of successful model retrieval from the server, Timeout still broadcasts the received weights, to help the convergence of the model of neighbors that might have failed to receive the current federated model.

This alternative behavior on bad network conditions eliminates the risk of learning slowdown caused by the server being heavily congested, because a lack of answer from the server will not prevent the nodes from learning together. Since for each node, the amount of neighbors is significantly lower than the amount of nodes in the experiment, the risk of network congestion at the working nodes is much lower.

Finally, the algorithm also defines "for free" the behavior for nodes that cannot reach any other node. Since awaiting the models from neighbors times out whenever neighbors are unreachable within a given amount of time, the node will simply receive no model. In this case, the aggregation step becomes trivial by simply returning the current model of the node. The model then keeps on learning locally and attempts communication again at the next round.

4.2. Results of the first benchmark

The first benchmark was run on a computer with a Ryzen 9 5950X processor with 16 physical cores and 2-to-1 SMT enabled. As a reminder, we ran the Timeout algorithm on 27 network setups (3 values for drop rate, 3 for throughput, 3 for delay) and we observed the time to run through 80 iterations. Figures 2-4 show the time elapsed between the first and last iterations as a function of RTT delay (x-axis) and maximum throughput (y-axis). Color is a function of height for better visual clarity. From these figures, we can observe some preliminary results.

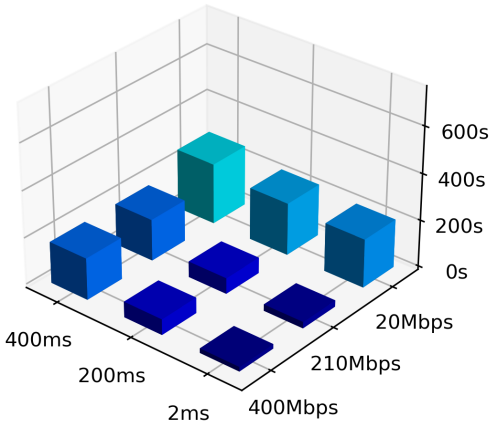

Figure 2: Benchmark 1: 0% drop rate

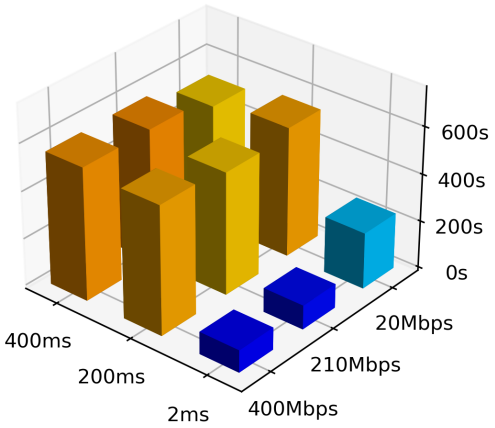

Figure 3: Benchmark 1: 2.5% drop rate

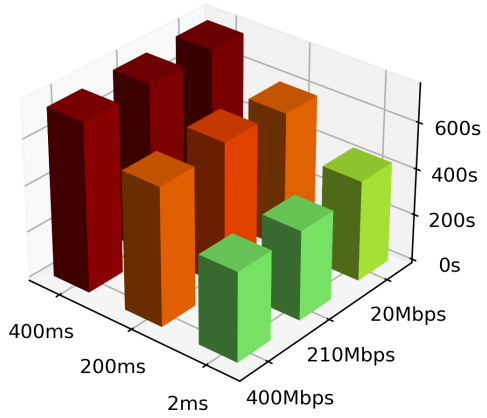

Figure 4: Benchmark 1: 5% drop rate

The largest influence on the running time of the Timeout algorithm clearly seems to be the packet drop rate. Losing 5% of the packets gives an average of 581 seconds to converge to the end of 80 iterations, while the same average computed over experiments with no drop rate gives an average running time of 137s, resulting in a reduction factor of 4.25. There may be several explanations for this phenomenon: the fact that some packets are dropped will clearly influence the running time, but this is exacerbated by the TCP congestion control algorithms that detect packet losses and attempt reducing the transmission rate to reduce load on the network. We did not experiment further with other transmission mechanisms, such as custom re-transmitting protocols based on UDP that do not suffer from an overly conservative congestion control mechanism, but it could be an interesting addition.

The second influence on the running time of the algorithm seems to be the round-trip time between two given nodes. By increasing the delay of a packet, we can delay iterations as they depend on sequential communication between nodes. We can however expect that the influence of this parameter has a lesser impact on the final running-time of the algorithm, since congestion algorithms are mostly unaffected by reasonable RTTs, and all values of delay were smaller than the timeout value used in the algorithm. To quantify this effect, we can observe that with a 400ms RTT, the average of experiments runtime was 496s, while in the same experiments with a 2ms RTT, the average runtime was 210s. This makes for a 2.36 reduction factor, which is observable on the graphs.

Perhaps surprisingly, throughput had less of an impact than the other two parameters. In particular, the difference in the execution time of the 210Mbps and 400Mbps experiments is hardly noticeable, which suggests that network throughput is no longer a bottleneck when it is higher than 210Mbps in this setup. Secondly, the reduction factor between the average of experiments with the highest and lowest value for throughput was only 1.19. This suggests that transfers made over the network are small, as small transfers (that fit in a few packets) tend to much more influenced by packet delay than by throughput, which matters more for large transfers. That explanation is reasonable, considering that information transmitted over the network mostly consists of neural network weights transfers, each having a size of a few hundred kilobytes.

These hints can be made more rigorous using a simple OLS regression. We used the 27 experiments as data points: The running time of the algorithm was modelled as the response variable, and the explanatory variables used for regression were the parameters of the network after standardization. The coefficients obtained are summarized in the following table:

Standardized Parameter	Coefficient
Drop rate	181.3530
Delay	116.9961
Throughput	-26.7155
Constant	375.8079

This tends to confirm our initial observations. With an R^2 value of 0.864, we can argue that the model fits well in the sense of least-squares, and corresponds to the prior visual analysis. It is also expected that the coefficient be negative for the throughput and positive for the others, as a high throughput is expected to reduce the running time, whereas high RTT and drop rate tend to augment it.

4.3. Second benchmark: the three algorithms

The second benchmark was run on the EPFL cluster. The results are therefore not directly comparable to those of the first benchmark.

4.3.1. No packet drops

Figure 5 shows the time elapsed of all three algorithms after each iteration, out of a total of 80. This first part of the benchmark had a packet drop rate of zero. It shows multiple important results:

In the world-net network setup, decentralized learning fared better than its peers by a large factor. We may explain this by reminding that decentralized learning is the only algorithm of the three that does not need the intervention of a federating server. In this scenario, that server is simulated as being located in Stockholm, meaning it might have been hard to reach for nodes simulating far-away cities.

In the bad-net network setup, we get similar results, except for the fact that the Timeout algorithm seems to behave more like the timeout one. This is likely due to the fact bad-net has a worse connectivity than world-net, worsening the capacity to reach the federating server. The fallback mechanism of the Timeout algorithm made the algorithm behave like its decentralized cousin. The same behavior happens in the mid-net experiment.

Finally, the best scenario shows a fairly similar time taken by all algorithms. Decentralized still behaves slightly better due to the lack of communication with the federated server, but the others have a comparable performance within a few seconds.

4.3.2. What about drop rate ?

The natural continuation of the previous set of experiments is to increase the drop rate to levels considered high (5%). Figure 6 describes the time elapsed when by each of the 80 iterations began. We then observe a fairly different behavior:

In the world-net setup, we see that the federated algorithm is outperformed by the decentralized one, in the same way as the 0% drop experiment. Still, Timeout becomes the best performer. This suggests that in the case of a particularly bad iteration where no communication is possible at all, Timeout skips both sharing the weights with the federating server and with its neighbors. It trains locally and retries communicating results at the next iteration, making it gain large amounts of time

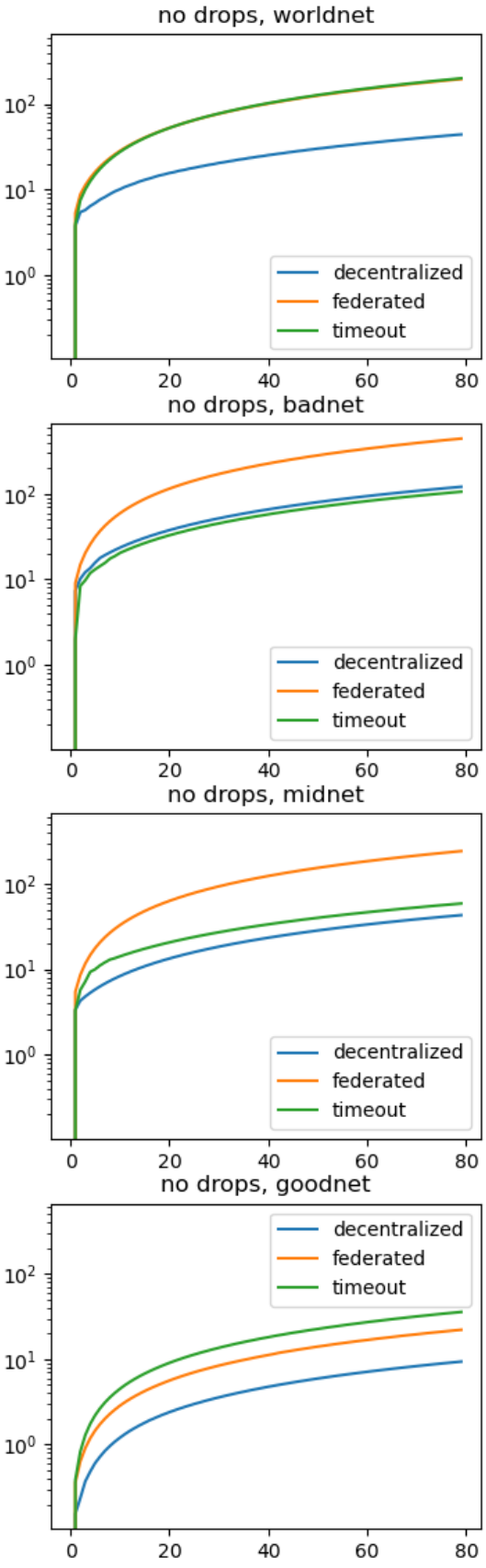

Figure 5: The time taken by all three algorithms to run 80 iterations without drops. Y-scale is logarithmic.

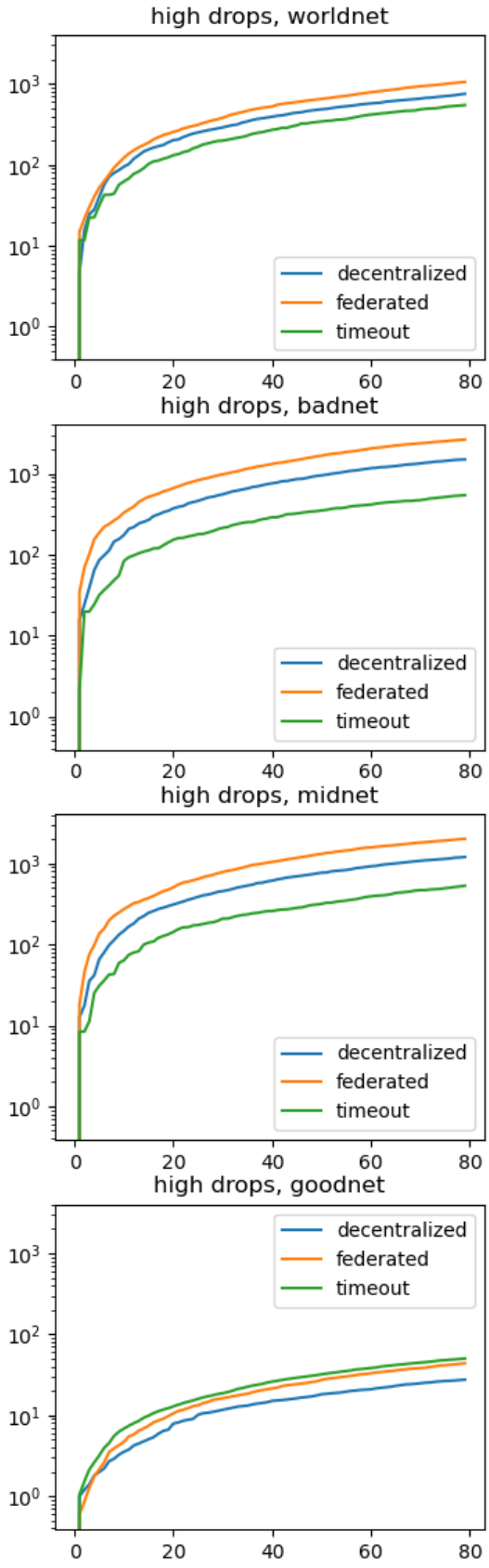

Figure 6: The time taken by all three algorithms to run 80 iterations with 5% drops. Y-scale is logarithmic.

in the process.

This is even more visible in the mid-net and good-net setup, where Timeout converges significantly faster than its peers. Again, this is due to its flexibility in handling situations when weight sharing is difficult. This does not mean that the algorithm only converges because it times out: in mid-net, about 74.4% of iterations successfully reached the federating server.

4.3.3. Massive drop rate

For the final part of this benchmark, we also recreated the same experiments with a 15% drop rate. The results are way simpler to interpret: within a running time of one hour, none of the decentralized and federated algorithm managed to run for a single iteration in any of the network setups, including good-net. This dramatic behavior is surprising, as a drop in 15% of the packets is not expected to cause a slowdown of training by a factor of well over 100 compared to the experiment with no packet drop. We blame this behavior on the TCP congestion control which was not built to withstand such a high packet drop rate, and severely limits the throughput of the outgoing link to attempt to ease the load over the network. While this phenomenon also occurs for Timeout, its non-blocking nature enabled it to ignore the network issues and keep learning. Figure 7 shows the time taken by Timeout in the four network setups.

Occasionally, in the world-net configuration, some weights reached their destination, still enabling the sharing of model information. Accounting for all four nodes, about 16.8% of iterations still managed to reach the federated server, enabling a reasonable amount of communication and learning.

Still, in the other setups, no node managed to contact another node, meaning that the training was done fully autonomously on every one of them. This explains the exact same graph repeated three times: the algorithm timed out on every single iteration, making the time taken per iteration constant.

The difference between the synthetic setups (good-, mid-, bad-net) and world-net is difficult to explain. While the curves look sim-

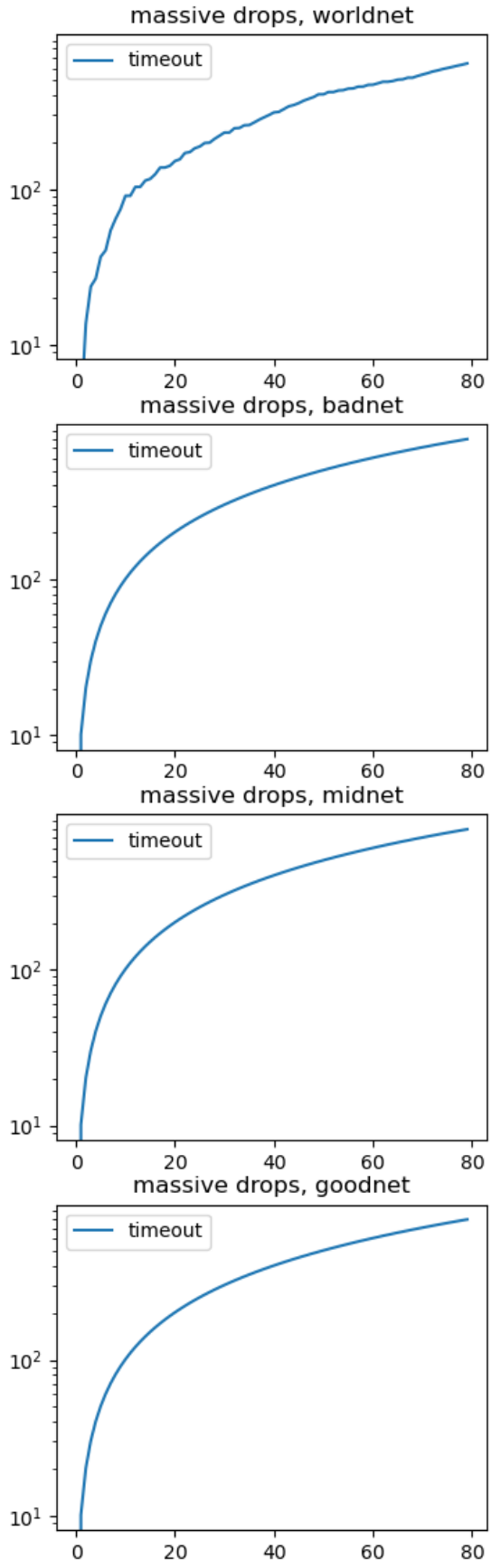

Figure 7: The time taken by Timeout to run 80 iterations with 15% drops. Y-scale is logarithmic.

ilar, some world-net iterations still reached the federating server and no good-net ones did, even though world-net has worse characteristics by every metric. A possible explanation for this behavior would be the fact that since world-net is asymmetrical, packets arrived at different time intervals at the federating server, which could respond faster as it was not processing other requests. Therefore, it may have avoided triggering the congestion control of worker nodes as much.

We can extract from this last experiment that a 15% packet drop rate is too high for regular algorithms to function properly. Non-blocking algorithms have a way better chance of running properly, assuming any kind of communication is possible in the first place.

4.4. Side note on accuracy

We ran the three algorithms on good-net with 5% packet drops for 200 iterations. We measured their accuracy at every iteration, obtaining the results described in Figure 8.

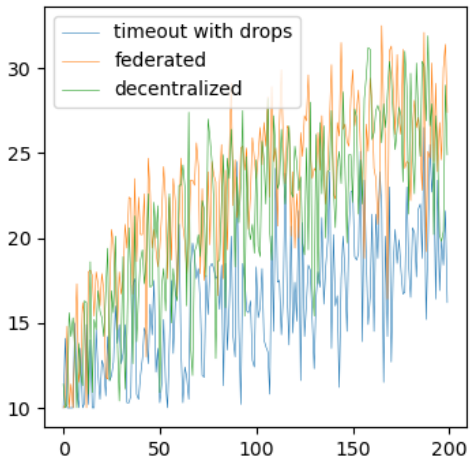

Figure 8: Accuracy per iteration in good-net with 5% packet drop

A (noisy) trend appears: all algorithms obtain an accuracy that grows over time, but that of Timeout grows at a slower pace. We believe this is due to the asynchronous implementation of the nodes. Since they are allowed to time out and enter new iterations without having received the information from other nodes (or at least not all of them), it follows

that they have to use models that have been trained on fewer data points. Another similar phenomenon might also occur: when the node receives late information from the federating server (in the sense that it has already timed out for that iteration), all the training that the worker node has performed is lost, as it will overwrite its own model with the old one received from the federating server. This leads to learning iterations that are completely lost in terms of training, as they are overwritten by older information from the server.

This experiment therefore suggests that one should pick carefully the kind of algorithm that one uses for decentralized learning: while the speed of execution of Timeout is especially attractive in hard network conditions, its accuracy is slower to grow than traditional algorithms because some data it deals with is either missing or stale. In poor network conditions, Timeout runs several times more iterations per unit of time than its counterparts, while also learning less per iteration. If the objective is to obtain the highest accuracy possible in the shortest amount of time, both philosophies may be viable and both must be evaluated.

5. Future work

There are several steps that may improve this study. We propose the following trails of research:

A good start would be to study why synchronous algorithms block so hard when exposed to very high amounts of packet drops. We blamed this on TCP congestion control systems, and that could be confirmed by investigating whether communication protocols with congestion control disabled perform better in these situations. The project of Emna Fendri comes to mind.

On another note, while we evaluated over several network parameters, there is one we left out: jitter. One could also evaluate whether this parameter has a measurable influence over the speed of convergence. This would add to the realism of simulations, but finding sources on typical values for Internet jitter might be hard.

6. Conclusion

In this paper, we hope to have established three things.

It is possible to implement an algorithm that combines federated and decentralized learning in a way that does not differ significantly from traditional collaborative learning methods, for example by reusing the abstractions implemented in tools like Decentralizepy.

We also found that the implementation we provide to this algorithm suffers the most when being exposed to a bad connection drop rate, as could be expected from the fact that it internally uses TCP, which artificially decreases network performance in the case of recurrent packet loss.

Finally, we found that its running time is similar to that of the implementations of decentralized and federated learning in high-quality networks, and especially outperforms them in hard network conditions that tend to favor non-blocking algorithms. In more reasonable setups, it mostly behaves like federated learning, which corresponds to what we expected. This must be put in perspective with the fact that in hard network conditions, Timeout deals with incomplete and stale information, which implies a lower classification performance compared to algorithms that would rather block until they have received the fully up-to-date models. Ultimately, the choice between both is up to the developer of the system.

Acknowledgements

The author would like to thank Rishi Sharma and Rafael Pires for supervising this research project. Their advice was of crucial importance. This work was conducted at the Scalable Computing Systems Laboratory, under the direction of Anne-Marie Kermarrec.

References

[1]A. Dhasade, A.-M. Kermarrec, R. Pires, R. Sharma, and M. Vujasinovic, «Decentralized learning made easy with decentralizepy», in Proceedings of the 3rd workshop on machine learning and systems (2023), pp. 34–41.

[2]P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, «Kollaps: decentralized and dynamic topology emulation», in Proceedings of the fifteenth european conference on computer systems, EuroSys '20 (2020), ISBN: 9781450368827, 10.1145/3342195.3387540.

[3]V. Gramoli, R. Guerraoui, A. Lebedev, C. Natoli, and G. Voron, «Diablo: a benchmark suite for blockchains», in Proceedings of the eighteenth european conference on computer systems (2023), pp. 540–556.

[4]H. Zhang, A. Elmokashfi, and P. Mohapatra, «Wifi and multiple interfaces: adequate for virtual reality?», in 2018 ieee 24th international conference on parallel and distributed systems (icpads) (IEEE, 2018), pp. 220–227.