Code was uploaded at `https://github.com/zhzhang2018/DecentralizedFL`

Start from section 3 for most recent updates. Section 1-2 was also updated - but they're more about finding the right parameters back in January when the whole thing just started.

I don't believe this document is sufficient for a publication. For example, the results and conclusions would be more convincing if it were run against a more complicated dataset (such as CIFAR and the Shakespeare dataset used in McMahan et al. 2017). In terms of organization, the document's layout combined methods, results, and analyses into single sections for every set of experiments, and the wording smelled like tutorials. It's probably better suited as a guide for future researchers, which they can use to determine what topic to pursue next, and how.

Would it work as an open-source arXiv document?

1 Federated Learning

1.1 Introduction

Federated Learning was an approach to learn on decentralized data, first proposed by McMahan et al. [1]. The idea is to have a central server send a model to a batch of randomly selected devices ("clients") with local data, perform a local update independently on each client, and then let the server aggregate all the updated client-side model parameters, and take their average as the model parameters in the next iteration. This procedure is named the "FedAvg" algorithm.

Ever since then, Federated Learning has become a rapidly growing research field. Many parts of the FedAvg algorithm could be adjusted for different situations, such as testing different aggregation methods (other than averaging), reduction of communication cost during aggregation, application considerations in real working environments, and so on. [Insert references here].

Switching from a centralized sharing scheme in FedAvg to a decentralized one could also allow models to reach a consensus at the global minimum, while avoiding communication bottleneck due to the server node. Lalitha et al. have adapted this decentralized scheme on Bayesian Network learning [2]. Savazzi et al. has implemented a decentralized flavor of FedAvg as well, but adapted for massive IoT networks [3]. In addition, Taya et al. implemented a decentralized training scheme that share the functional instead of the parameters [4].

1.2 Decentralized Federated Learning

We attempt to solve the MNIST classification problem [5] with a general decentralized federated learning scheme, outlined in this section. For the following, we assume that G is an undirected graph that describes client connectivity, and maintains strong connectivity over time. $\mathcal{N}_i(t)$ represents the nodes connected to client i at time t.

We use K to represent the total number of clients, and $w^i(t)$ to represent the i-th client's parameters at time t. The pseudocode is as follows:

1. Initialize all clients with the same parameters $w(0)$.
2. **for** $t = 1, 2, \cdots, T$ **do:**
3. **for** $i = 1, \cdots, K$ **in parallel do:**
4. **for** $j = 1, \cdots, |D_i|E/B$ **do:**
5. $w^i(t) \leftarrow w^i(t) - \eta \nabla l(w^i(t); b_j)$**;**
6. **for** $k = 1, \cdots, s_\epsilon(t)$ **in parallel do:**
7. **for** $i = 1, \cdots, K$ **do:**
8. $w^i(t) \leftarrow w^i(t) + \sum_{j \in \mathcal{N}_i} \alpha_{ij}(w^j(t) - w^i(t))$
9. $w(t+1) \leftarrow w(t)$

where η, α_{ij} are constants that determine update rates, D_i is the local dataset for client i, B is the batch size for each client's local updates, b_j is the samples drawn in the j-th batch, and l is the loss function. The value of α_{ij} could be different according to i and j in the implementation.

Notice that step 8 follows the consensus protocol, but might not be really necessary. If each client is able to receive the full weight parameters from all neighbors in this step, then we might as well just get to the consensus position as fast as possible instead:

$$8.\ \ w^i(t) \leftarrow \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i \text{ or } j=i} w^j(t)$$

This algorithm is not a generalization to the centralized FedAvg algorithm. The FedAvg algorithm effectively uses a star graph G_S with $K+1$ nodes, where the server node is the center. Lines 1-5 would still apply because the server node does not have its own local data. Lines 6-8 would be slightly different, in that the server does not put its own model in the averaging, and no client participates in the averaging, either. The FedAvg update rules would be like:

$$6. w^S(t) \leftarrow \frac{1}{K * C} \sum_{j \in C(t)} w^j(t)$$

7.Select a new set of clients $C(t+1)$

8.for $i \in C(t+1)$ do :

9. $\quad w^i(t+1) \leftarrow w^S(t)$

The organization is much easier in this case. In addition, in FedAvg, the server only selects a subset of all the clients for each round. It can be proven that the aggregation algorithm always converges when the weight $\frac{1}{KC}$ is smaller than the number of clients participating in the averaging.

Observe that one key property of FedAvg is that all the clients start from the same model before starting each round of local training. The trained model could risk diverging if the starting weights are too far away from each other. This serves as an important thing to preserve when designing the parameters for the decentralized FL.

1.3 Overview of the Report

In the following sections, we discuss parts of the Federated Learning algorithm that could be adjusted, and conduct experiments to see their effects. The report uses the MNIST classification as a benchmark for testing. Section 2 introduces hyperparameters regarding training, and their choices. Section 3 introduces different ways that Federated Learning algorithms could be tweaked. For example, we observe that:

- Clients could share a small portion of training data with each other to stabilize training, especially if the training data is skewed between clients;
- Models could still fit the training data with varying normalization values and/or loss functions, provided that they are not too off;
- The parameter sharing step of Federated Learning could be carried out by breaking the model into multiple segments, in order to reduce communication cost.

1.4 Experiments to Establish Baselines

There are several values in the decentralized FedAvg process that should be important to the training performance:

- The number of iterations to execute the consensus protocol so that clients reach the same model weights before starting another round of local update.
  Use s to represent this value. If s is too small, then the client network would not be able to reach a consensus before making another round of update, and the model may diverge. This parameter is also adaptable, meaning that we can set it as $s_\epsilon(t)$ where s_ϵ means the least amount of iterations it would need for the entire network's clients to reach consensus with error at most ϵ between any pair's parameters.
- The amount of update steps done throughout one epoch, in-between the model aggregation steps.
  We use E to represent the number of epochs for this. For example, if $E = 3$, then

each client goes through its local dataset for 3 times before sharing their model parameters. If $E = 0.5$, then it only goes through half of the training dataset before aggregation.

- Network density (potentially time-varying).
  The network density can be described by the second eigenvalue of its Laplacian. A denser network has more connections, and thus would have faster convergence rate for the consensus protocol. This is covered in detail in 3.2.1.

- .

We would test with those values to determine a reasonable configuration in the following section. In addition, the model construction and the training environment and setup could also affect the performance.

1.5 Methods

The network setup is based on the official PyTorch example [6].The network consists of a 32-channel convolution layer, followed by another 64-channel convolution layer, both with a kernel size of 3 and followed by an activation of ReLU. After a 2×2 MaxPooling layer and a Dropout layer with probability 0.25 is a fully connected layer with 128 output values and ReLU activation. This is followed by another Dropout layer with probability 0.5, and then a second fully connected layer with 10 outputs. The outputs are turned into predictions for 10 classes through Softmax, and evaluated with negative log likelihood function against the correct label.

While the preliminary experiments in the following section were run on a local computer in Jupyter Notebook with a Python 3.8 kernel, the experiments in section 3 were run on the PACE cluster [7].

2 Effects of Training Setups and Hyperparameters in Decentralized Federated Learning

There are many different parameters that would affect the experiments.This section explores some of those methods, analyzes their performances, and points to some interesting discoveries.

2.1 Preliminary Testing with Decentralized Federated Learning

We first trained a single model with the entire MNIST training dataset ($|D| \triangleq 60000$ samples), as the baseline. The model reached 98.28% accuracy on the test set (6000 samples) after the first epoch.

Next, we ran the centralized FedAvg algorithm from [1]. We followed the parameters et in the original paper, where $K = 20$ clients and a central server train over the entire dataset. The dataset was randomly split and distributed across all clients IID and disjoint. In addition, each epoch only updates $C = 0.5$ of all the clients, meaning that $CK = 10$ clients participate each round of local model updates and parameter aggregation. To take care of the limited computational resources, and to reveal more interesting training behaviors across epochs, we reduced E from above 10 in the original paper to the range between 0.01 and 1. Batch size was also chosen to be $B = 10$.

The results confirmed that the parameter choice could produce decent models. Server side model reached 98% accuracy after around 20 rounds. Considering that there are $K = 20$ clients, $CK|D_1|E = CK|D|E/K = 0.005|D|$ samples were used to update in each round, and 10% of all samples were used in this experiment to achieve 98% accuracy. After 100 rounds, the number of samples processed would roughly equal to the size of the entire dataset, and the resulting test accuracy was a bit better than 98.28% in baseline.

Finally, we implemented decentralized federated learning (DFL), where we chose $K = 10$ as a first step, so that the number of clients sharing the parameters each round would be the same as the CK value in the original Federated Learning paper. The hyperparameters E and s were effectively set as $E = 1/200$ and $s = 100$, and we switched to $B = 64$ for faster training. The dataset was partitioned evenly into K local sections for the clients, in an independent and identical distribution manner. The accuracy history on the test dataset after each training epoch t was obtained by asking the client with the lowest loss $l(w^i(t); b_j)$ to output predictions. When the models were trained with completely random initial weights, the models quickly diverged, as the test accuracy quickly dropped to around 11.35%, no better than random guesses. At the same time, the loss history for each client increased (exponentially?) with t, due to the unbounded nature of negative log-likelihood losses.

In comparison, the training with partitioned dataset and uniform initial weights has been more successful. We tested with various E values, while using the same setup for the rest of the parameters.

When $E = 1$, the partitioned version reached an accuracy of 98.8% in the 38th epoch. Each time effectively passed through 3.8 times of samples than in the entire dataset.
When $E = 0.05$, the test accuracy lingered around at 97% after 40 epochs of training ($40 \times 0.05 = 2$ times the dataset size), and settled at 97.85% near the 500th epoch. The top-performing client successfully found a working set of parameters fast, but this result is not as impressive as the single-model control group, indicating that the model might have reached a local minimum instead.
When $E = 0.005$, we observed that the resulting test accuracy was smaller than 90%, and that the training loss started increasing after the 5th epoch, indicating an overfit behavior. Thus, we decided that $E = 0.05$ should be close to a sweet-spot for model aggregation frequency.

2.2 Effect of faster model aggregation frequency

In the standard Federated Learning setup, each client is supposed to have disjoint training datasets, for privacy concerns. Each training data should only be available to the client that hosts it, and the client is only supposed to share its model parameters without making the local sample somehow traceable by other clients. However, in an experiment setting, we could disregard this requirement, grant all clients access to the full or most part of the dataset, so that we can test the robustness of the simple averaging aggregation step. During one simulation, we had an interesting observation: None of the client models was able to learn a good classification model when the initial model parameters happened to be completely random.

The results below used the same setup as in the previous section, except for altering the value of E and giving full access to the MNIST training dataset to all clients.

If $E = 1$, meaning that in each epoch, all clients went through the entire dataset on their own (with different initial weights, of course), then the aggregated model's test accuracy reached 10.28% starting from the second epoch. The training loss history would reduce during the first epoch's training, but then suddenly reach a plateau after the first consensus aggregation.
If $E = 0.05$, meaning that each client went through 5% of the total samples at each epoch, and around the 50% of the training dataset as a group effort per epoch, then the final test accuracy for the aggregated model stabilized around 18% after a few epochs. The training loss during the process would decrease, but not by a lot.
If $E = 0.005$, then the setup was able to reach a test accuracy at around 86% after around 15 epochs. This means that the total amount of samples that the clients processed as a group would be only $KE = 5\%$ per epoch, and 75% up until the 15th epoch. Further training did not improve the test accuracy any further. Interestingly, in this experiment, the training loss history starts climbing down at the second epoch, starts to obviously dwindle down after the 10th epoch, and starts fluctuationg around a certain value at the 100th epoch. According to the test loss history, the point of overfitting happens at the 10th epoch.

The most possible explanation is that, when the aggregation became more frequent, the clients' models would become less different after each epoch, and the model parameters would reach consensus faster. When $E = 0.005$, the models reached consensus at a suboptimal point first, and then could not advance forward. When $E = 0.05$ and $E = 1$, the models trained for too long before aggregation, and the resulting centroid point was a bad-performing set of parameters.

The models were supposed to train better with more data available, especially with access to the entire dataset. Thus, the experiments above indicate that the federated learning method is fragile without uniform weights across clients. In the following sections, we shall make sure all models start with a uniform set of weights, unless otherwise mentioned.

3 Variations on training setups, and outcomes

In the previous section, we have shown the effects of hyperparameters on the federated learning when training on the MNIST dataset. In this section, we mostly settle down on a fixed combination of the hyperparameters, and instead investigate the model behavior with different optimization setups. Note that MNIST is empirically easy to train with CNNs [citation needed], so some of the results and observations need additional verification using the CIFAR-10 dataset. (the painful process of tuning CIFAR started on 0305-0312 page 12-20 with shallow network, and then 0312 onwards (page 3) for slightly deeper PyTorch network.)

3.1 Different dataset distributions

3.1.1 Different dataset sizes

To create clients with different dataset sizes, we drew samples from a Gaussian distribution $x_{1:K} \sim \mathcal{N}(0, \sigma^2)$, and then randomly and uniformly draw N_i samples for client i as its training data using the following equation:

$$N_i = \min\left(\frac{|x_i|N}{\sum_{j=1}^{K}|x_j|}, B\right)$$

where $N = 60000$ is the total number of samples in the MNIST dataset. The resulting dataset distribution for each client is thus iid. This was tested with different combinations of parameters as shown in table below. Define e_p as the number of epoch where the overall accuracy first exceeded $p\%$, and the preliminary results were as follows:

	$K = 10, \sigma^2 = 1$	$K = 30, \sigma^2 = 2$	$K = 50, \sigma^2 = 2$	$K = 30, \sigma^2 = 10$
e_{90}	6	6	10	8
e_{97}	80	36	52	51

Note that, when $\sigma^2 = 10$, the variance is high enough for up to 50% of the clients to only have $B = 64$ data samples to start with. Most data were concentrated in a selected few clients. This unbalanced distribution facilitated the learning of these few with more training data, but their contributions were weakened during parameter sharing because they have the same weight as the ones with fewer training data. This resulted in slower training when σ^2 is higher, as expected. One way to overcome this is to use a weighted average sum, where the weight w_i for the parameters of client i is proportional to N_i, as proposed by Hu et al. [8].

3.1.2 Partially Overlapping Training Data

In addition to the unbalanced partitioning in the previous subsection (3.1.1), we tested to see if sharing certain data between clients would help. We introduce the parameter

$S \in [0, 1]$ to describe the amount that are shared. Each client would train on N_i' training samples where:

$$N_i' = N_i + S \sum_{j=1, j \neq i}^{K} N_j$$

The shared samples are drawn uniformly, and not necessarily the same for different clients. Each client i would share a total of SN_i training samples to other clients, giving $\frac{SN_i}{K-1}$ to each.

3.1.3 Skewed Training Data

Wang et al. have already investigated the effect of skewed datasets on federated learning [9]. In their paper, they partitioned the CIFAR-10 dataset, so that each client only has access to samples of the same label. For conveniency, we denote the dataset partition as:

$$D = \bigcup_{m=1}^{10} D_m; \quad \forall m \neq n, D_m \cap D_n = \varnothing$$

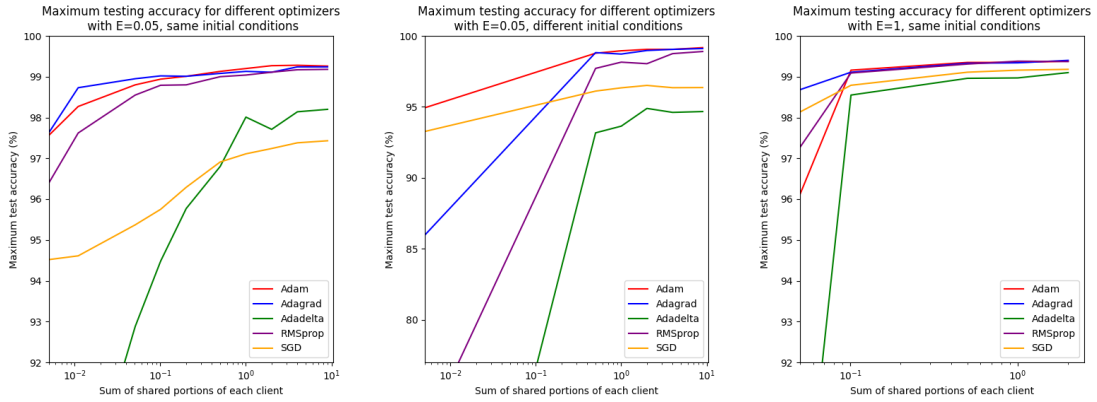
where D_m is the subset containing all data pairs with output labels as m. During partition, each client i is assigned a dataset $D^{(i)} \subset D_{c_i}$, where integer $c_i \in [1, 10]$ indicates a specific label. As demonstrated in [9], this assignment led to poor performance on CIFAR-10, and could be alleviated by pre-allocating a shared global dataset containing uniform label distribution. Running the similar training procedure on MNIST gave different results: The global model converges to an accuracy above 90%, but is relatively poorer compared to the models trained with less skewed datasets.

In the following, we compare the results using different optimizers provided by PyTorch with empirically-determined learning rates on a completely skewed dataset, with each label distributed to exactly one of the $K = 10$ clients. In addition, we investigate how sharing only a small percentage of the data samples can greatly alleviate the performance reduction. The learning rates (Table 1) were determined empirically to stabilize training and maximize performance, while keeping the training reasonably fast.

Table 1: Empirical learning rates for each optimizer

Optimizer	Adam	Adadelta	Adagrad	RMSprop	SGD
Learning rate	0.001	1	0.001	0.0005	0.01

Figure 1(a) shows the average model performances with S ranging from 0 (no data shared) to 1 (the entire dataset is shared - equivalent to training all clients with the same dataset). The performance was determined by the clients' models' average accuracy on the test dataset after 500 epochs, with $E = 0.05$ indicating that each client trains on its



(a) $E = 0.05$, same initial weights (b) $E = 0.05$, different random initial model weights (c) $E = 1$, same initial weights

Figure 1: Comparisons of maximum model test accuracies (%) with different values of S under different optimizers and training setups. Note that the horizontal axis is the sum of shared portions, meaning that a value of 2 indicates each client has the parameter $S = 2/K = 0.2$.

entire available dataset after every 20 epochs. The figure indicates that performances are dependent on optimizers, and more importantly, the top performances did not differ by too much, as long as $S > 1\%$, corresponding to 10^{-1} on the horizontal axis. Further tests on models without same initial weights in Figure 1(b) for $S \geq 5\%$ confirmed its stability with the three better optimizers; however, experiments with $S = 0$ showed that even those optimizers could potentially lead to bad models, which was not the case with same initial weights.

To investigate whether the observed convergence has anything to do with the more frequent updates, we repeat the experiments using $E = 1$, for 50 epochs instead of 500, and with same initial model weights. The results are similar to having $E = 0.05$, as shown in Figure 1(c).

At this point, one may question what exactly is the difference between the effects caused by sharing data samples and by using less skewed data. We now try to combine both ideas and, for simplicity, we again use $K = 10$ clients, each assigned a data subset $D^{(i)}$, where the majority of the training data belongs to the dataset D_i that only has label in category $i, i \in [1, 10]$. Let:

$$U_i \triangleq \frac{|D_i \cap D^{(i)}|}{|D_i|}$$

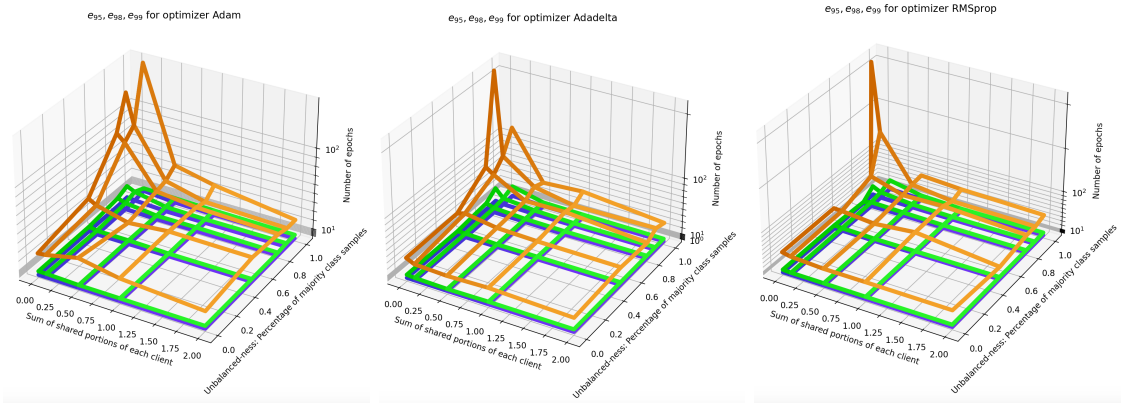
This value defines the "skewed-ness" of i's training data, or the percentage of the majority data in the overall dataset that are assigned to client i. The experiments in Figure 1 had

$U_i = 1$, for example.

In the following experiments, we assume $U_i = U$ for all i. For $S = 0$, each client's training data $D^{(i)}$ is disjoint from other clients' data, and contains $U|D_i|$ samples of the overall dataset subset D_i with label i, as well as $(1 - U)|D_i|$ data samples from the other labels. For $S > 0$ experiments, we would follow the same procedure for obtaining $D^{(i)}$, and after this, each client shares $S|D^{(i)}|$ samples to others, and receives other clients' data. In the end, an enlarged dataset $D'^{(i)}$ is formed and used in the training, with size $|D'^{(i)}| = (1 + S)|D^{(i)}|$.

We trained the models with around 300 epochs and with $E = 1$ for a more practical update frequency. In addition, to better portrait the experiment behavior, we recorded the characteristic epochs e_{95}, e_{98}, e_{99}, which are the first epochs where the model's best prediction accuracy on the testing set would reach 95%, 98%, and 99% respectively. We tested with the top-performing optimizers (Adam, Adadelta, RMSprop) with same learning rates from Table 1, had the client models start with uniform initial weights, and kept other parameters the same. The results are shown in Figure 2 and 3.

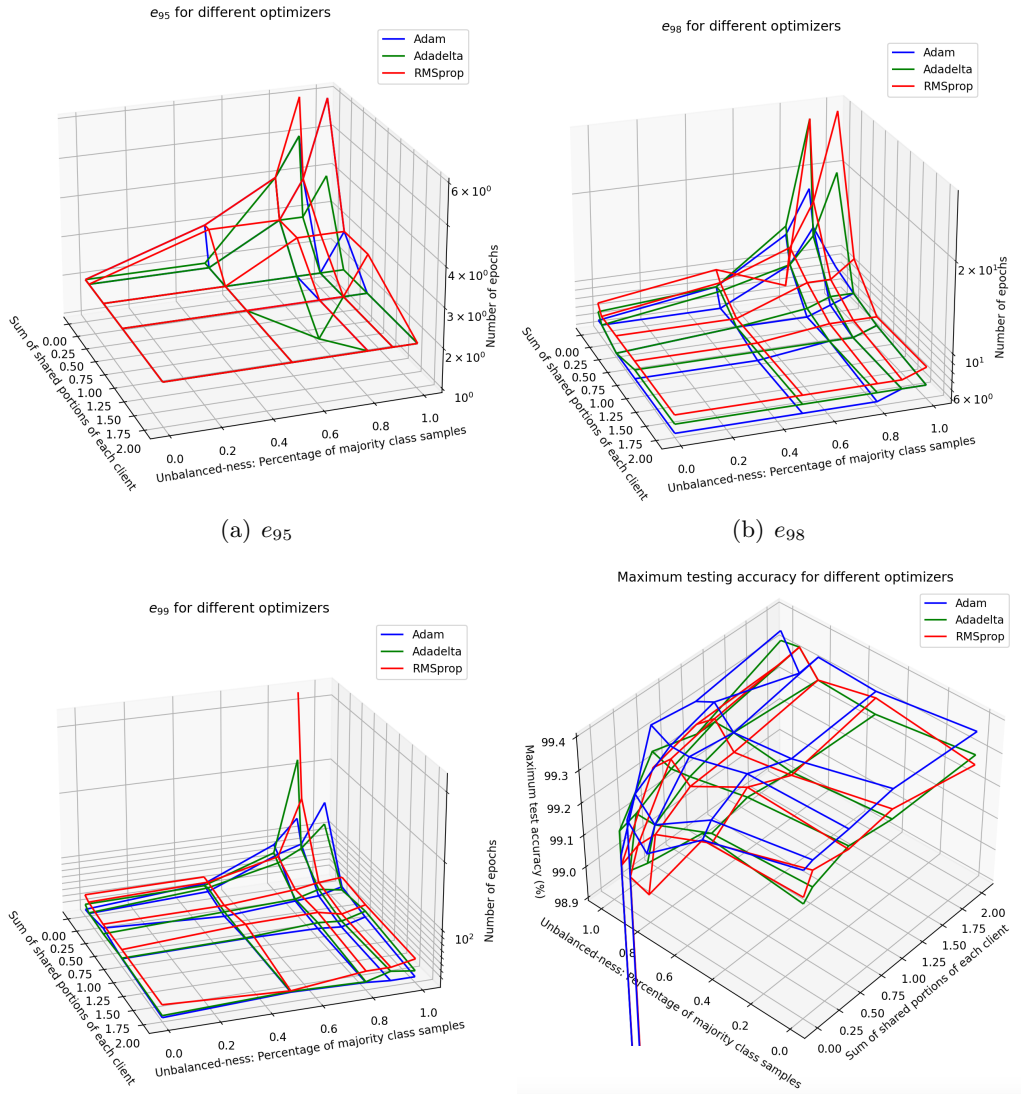
The general trend, as observed in Figure 2, is that the number of epochs it takes for the client models to reach a certain accuracy is larger if S is smaller or if U_i is larger, illustrated most obviously by e_{99}. The shape of e_{98} and e_{95} has much less variation, but the trend is still visible. Figure 3(a-c) plots the same data, but in a different view that compares optimizers. In addition, Figure 3(d) plots the max average model accuracies, which mostly stay around 99% regardless of S and U, except for when $S = 0$ and $U = 1$, the condition that failed to produce good models in [9].



(a) Adam (b) Adadelta (c) RMSprop

Figure 2: Comparisons of e_{95}(blue),e_{98}(breen),e_{99}(orange), with different values of S and U under different optimizers. Missing points indicate that the corresponding accuracy was never reached before the final epoch.

It should be pointed out that other researchers have also come up with different ideas for



(a) e_{95} (b) e_{98}

(c) e_{99} (d) Max performance

Figure 3: Comparisons of different optimizers' performances with different values of S and U at thresholdse_{95}, e_{98}, e_{99}, as well as best performance accuracies. Missing points in (a-c) indicate that the corresponding accuracy was never reached before the final epoch. (d) is cropped to exclude values too low.

reducing variance caused by skewed datasets, such as the SCAFFOLD algorithm proposed by Karimireddy et al. [10].

3.1.4 Differently Normalized Training Data

While we need more work to understand the effect of data normalization to the general performance of the training, we suppose it's safe to assume that applied federated learning might have advantages in using different normalization schemes across devices. Thus, we investigate if the training could be successful with different normalization schemes on the MNIST dataset.

In PyTorch's example, the MNIST dataset inputs are normalized using the collective mean μ_M and variance σ_M computed from the entire training dataset [6]. We devise several experiments to check the effect of altering those normalization parameters.

Let μ_i indicate the normalization mean each client uses on its raw data. When shifting the normalization mean, each client's mean would be:

$$\mu_{1:4} = \mu_M - 0.1,\ \mu_{5:8} = \mu_M + 0.1,\ \mu_{9:10} = \mu_M + 0.3$$

Figure 4(a-d) reports the model test accuracy histories in this setting, and clients 9-10 with normalization parameters $\mu_{9:10}$ further from μ_M showed worse performances and could suffer from overfitting.

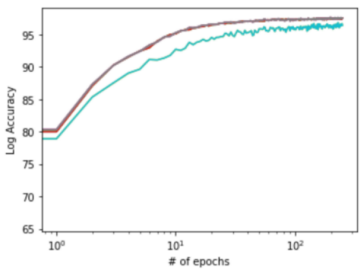
Further increasing the difference in mean values provided more drastic outcomes, as shown in Figure 4(e-f). This indicates that, while bad normalization parameters could hurt the performances (as shown by client 9 and 10), the majority of the clients' models are robust against this with averaging.

In addition, this training scheme could be combined with skewed dataset. As a first test, we assign D_i to client i for $i = 1, ..., K = 10$, and calculate the normalization parameters μ_i based on the mean and variance of the dataset D_i. After training with Adam as the optimizer with learning rate at 0.001 for 50 epochs, the global model reached an accuracy of at most 98% when the data sharing percentage is $S = 5\%$, and at most 86% when $S = 0$, indicating that model behaviors are relatively unaffected by small variation on normalization means.

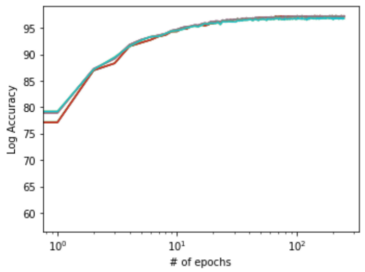
Figure 5 presents the results. The values when $S = 0$ were obtained from the last 10 epochs of training over 250 epochs, because the max accuracy histories were noisy, and displayed large shifts. With $S \geq 0.1$, the globally-shared data samples were helpful in stabilizing the training.

3.2 Different Sharing Methods

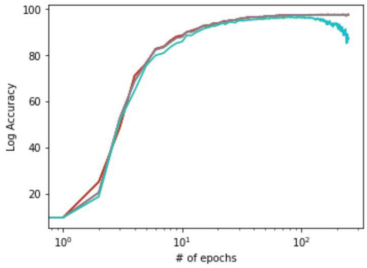
One thing in common for previous experiments is that they have to share the full set of weights each epoch. This can be costly in communication, especially in edge computing and other application scenarios with many different devices. Thus, some researchers are curious if it's feasible to reduce communication cost by only sharing a segment of the entire model to a selected few neighbors.



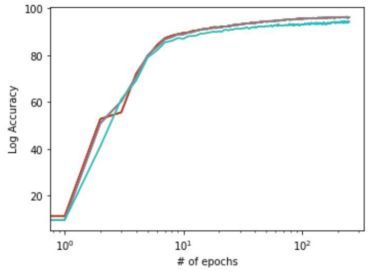
(a) Adam, $S = 0$, same initial weights. Max accuracy 97.64%.



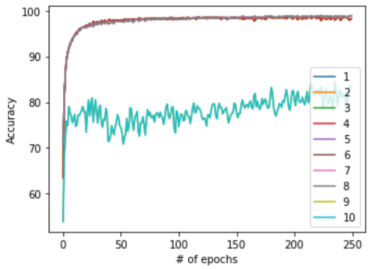
(b) Adadelta, $S = 0$, same initial weights. Max accuracy 97.34%.



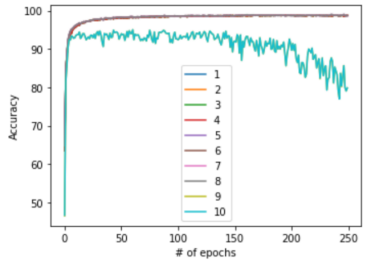
(c) Adadelta, $S = 0$, different initial weights. Max accuracy 97.55%.



(d) Adam, $S = 4$, different initial weights. Max accuracy 96.49%.

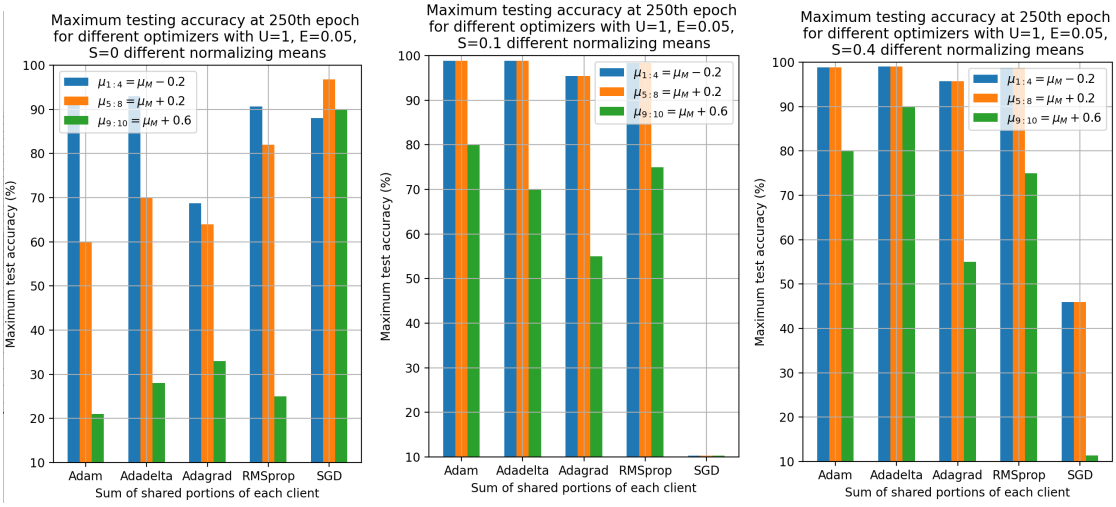


(e) Adam, $S = 0$, same initial weights. Max accuracy 98.96%.



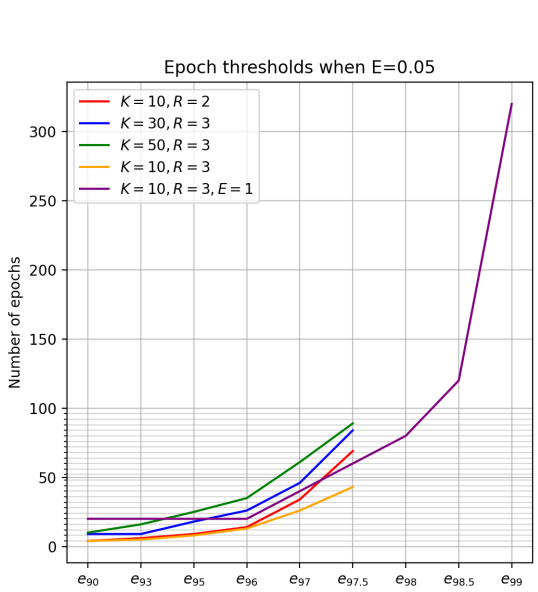
(f) Adadelta, $S = 0$, same initial weights. Max accuracy 98.97%.

Figure 4: Comparisons of test accuracy histories for clients with different normalizations. In (a-d), clients were trained with normalization parameters $\mu_{1:4} = \mu_M - 0.1, \mu_{5:8} = \mu_M + 0.1$,and $\mu_{9:10} = \mu_M + 0.3$. The histories of client 1-8 overlap as the brown line, while those of client 9-10 overlap as the cyan line. In (e-f), the normalization parameters are $\mu_{1:4} = \mu_M - 0.2, \mu_{5:8} = \mu_M + 0.2$,and $\mu_{9:10} = \mu_M + 0.6$, and the figures contained legends for clients.

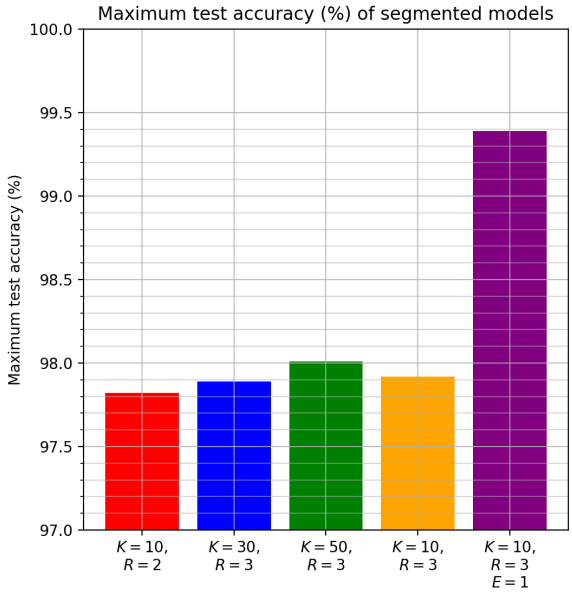


(a) $S = 0$ (b) $S = 0.1$ (c) $S = 0.4$

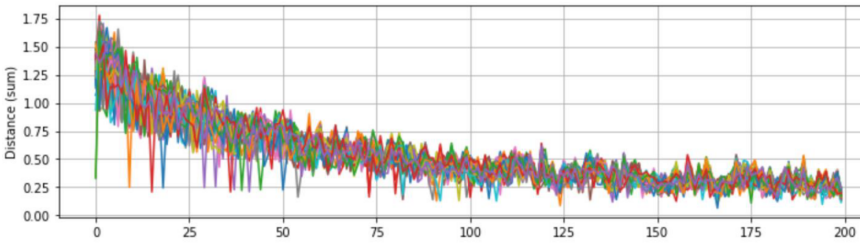
Figure 5: Accuracies for training under skewed dataset with normalization parameters $\mu_{1:4} = \mu_M - 0.2, \mu_{5:8} = \mu_M + 0.2$,and $\mu_{9:10} = \mu_M + 0.6$. Each presented value was the peak accuracy at the final 10 epochs of a 250-epoch training process.



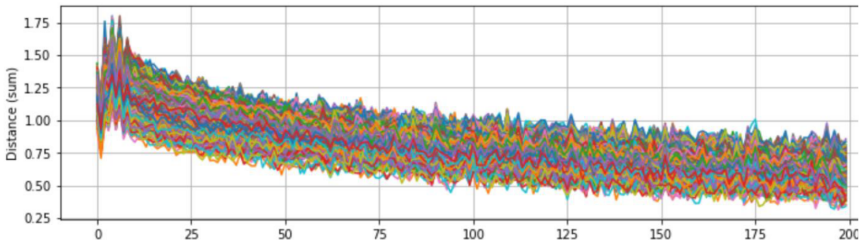
(a) Epoch counts



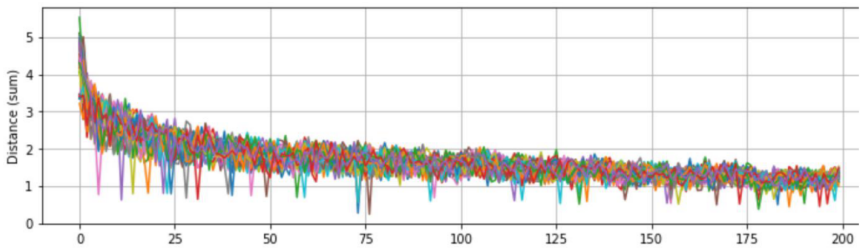
(b) Max accuracies



(c) Pairwise model distances for $K = 10, R = 3$



(d) Pairwise model distances for $K = 50, R = 3$



(e) Pairwise model distances for $K = 10, R = 3, E = 1$

Figure 6: (a-b): Epoch threshold and maximum accuracies of segmented federated learning with layer-wise sharing. All tested with $E = 0.05$ unless otherwise specified. In (a), the $E = 1$ data was obtained by multiplying the original data with $20 = \frac{1}{0.05}$. (c-e) History of every possible pair of clients' model parameter distance, using ℓ_2 norm, over 200 epochs.

3.2.1 Segmented Federated Learning

Hu et al. tested this idea by dividing the entire model W_i of a client i into several segments $W_i^1, ..., W_i^M$[8]. In the aggregation part, each client polls segments from a fixed set of R other clients, and performs weighted averaging according to each client's dataset size. This procedure could be done in a gossip approach, to save communication cost while still giving convergence. However, it is unclear how they divided the model into "segments".

As a proof-of-concept, we treated each layer as a segment, and randomly picked R clients out of the K clients for each segment. The training setup was the same as previous default setups (same initial weights, Adadelta optimizer, $S = 0, U = 0$), and the results after 200 epochs is shown in Figure 6. Note that for larger K, it becomes more likely that the resulting communication graph (i.e. who receives from who) for each layer segment is not fully connected, thus preventing a full model-wise consensus in the experiment.

The models could successfully converge to a set of parameters with sufficiently good performance with accuracy above 97%, as shown in Figure 6; in addition, having a denser graph (by having smaller K and/or larger R) requires less training epochs to reach an accuracy threshold. In comparison, however, the resulting model is worse than when each client trains on their own before sharing, as illustrated in Figure 6(b): The experiment with $E = 1$ shares less frequently, and reached an accuracy of over 99% in roughly equivalent to 320 epochs with $E = 0.05$.

The fact that the resulting models in different clients have non-zero distances in-between (Figure 6(c-e)) reveals that the models don't have to be the same to get good performance. For example, as [11] has observed, networks with permutations in parameters could give exactly the same performance.

However, there doesn't seem to be a theoretical guarantee that methods like this would always work? At each epoch, the consensus aggregation for each segment is based on a directed graph topology, and the worst case topology could (?) lead to oscillation or even divergence. Empirically, however, this seems to be a reasonable approach.

A. Segment Units

The model used under current study, a simple convoluted neural network (CNN), can be described by a list of tensors with dimension up to 4D. Denote each layer as W_{L_i}, so a CNN with l layers would have its weight $W = [W_{L_1}, ..., W_{L_l}]$ where $W_{L_i} \in \mathbb{R}^{C_{i-1} \times C_i \times w_i \times h_i}$ if the i-th layer is a convolution layer, $W_{L_i} \in \mathbb{R}^{(C_{i-1} \cdot I) \times C_i}$ if the i-th layer connects to a previous convolution layer where the image output would have size I, and $W_{L_i} \in \mathbb{R}^{C_{i-1} \times C_i}$. Here, we use C_{i-1} as the input channel size, C_i as the output channel size, and $w_i \times h_i$ as the convolution kernel size for layer i.

It thus appears that we can intuitively divide each layer's weights into basic units that can be shared between clients. Starting from the smallest granularity, the segmentation unit being shared can be as small as one parameter inside a convolution kernel. Larger, it could be one single convolution kernel $k(i, c_{i-1}, c_i) \in \mathbb{R}^{w \times h}$ that is taken from the i-th layer, and

corresponds to the weights from the c_{i-1}-th channel out of the C_{i-1} input channels and the c_i-th channel out of the C_i output channels. Even larger, it could be the weights $W_{L_{i,out},c_i} \triangleq [k(i,1,c_i),...,k(i,C_{i-1},c_i)] \in \mathbb{R}^{C_{i-1}\times w\times h}$ of the i-th layer that weighs the c_i-th output channel out of the C_i channels of the layer, or $W_{L_{i,in},c_{i-1}} \triangleq [k(i,c_{i-1},1),...,k(i,c_{i-1},C_i)] \in \mathbb{R}^{C_i\times w\times h}$ that handles the c_{i-1}-th input channel. Even larger, the entire layer W_{L_i} could be a unit. Going further, the largest possible unit is the entire model's weights, which is what the standard federated learning algorithm does.
To summarize the notations: For some client j, its model weights are as follows:

$$\begin{aligned}
\text{Full model } W^j &= [W^j_{L_1},...,W^j_{L_l}] \text{ for client } j, \text{ with the number of layers being } l;\\
\text{Single layer } W^j_{L_i} &= [W^j_{L_{i,out},c_1},...,W^j_{L_{i,out},c_{C_i}}], \text{ aggregated by output channels, or}\\
&= [W^j_{L_{i,in},c_1},...,W^j_{L_{i,in},c_{C_{i-1}}}]^T, \text{ aggregated by input channels;}\\
W^j_{L_{i,out},c_i} &= [k^j(i,1,c_i),...,k^j(i,C_{i-1},c_i)], \text{ with kernels for all input}\\
&\text{channels corresponding to the } c_i\text{-th output channel,} 1\le c_i\le C_i,\\
W^j_{L_{i,in},c_{i-1}} &= [k^j(i,c_{i-1},1),...,k^j(i,c_{i-1},C_i)], \text{ with kernels for all output}\\
&\text{channels corresponding to the } c_{i-1}\text{-th input channel,} 1\le c_{i-1}\le C_{i-1};\\
k^j(i,c_{i-1},c_i) &\in R^{w\times h} \text{ is a convolution kernel at the i-th layer between two channels.}
\end{aligned}$$

Assuming that $k^j(t)$ is one instance of the smallest sharing unit (chosen from above) of the j-th client at the t-th iteration, the algorithm proposed by Hu et al.[8] would either predetermine or randomly determine a set of other clients $j^t_1,...,j^t_R$ on-the-go, and poll the model weights using:

$$k^j(t+1) = \frac{|D^{(j)}|}{|D(j,t)|} + \sum_{m=1}^{R}\frac{|D^{(j^t_m)}|k^{j^t_m}(t)}{|D(j,t)|} \text{ where } |D(j,t)| = |D^{(j)}| + \sum_{m=1}^{R}|D^{(j^t_m)}|$$

(Needs stability analysis)

In other words, the FedAvg algorithm is carried out in segment-levels, instead of in terms of full models

B. Connected graphs and small-world phenomenon

From a practical point of view, the original FedAvg algorithm requires using a central client to aggregate the models, and this could become a bottleneck in terms of performance. Because the result of FedAvg is updating local model parameters to the average value, this aggregation process can be replaced by a consensus algorithm, which achieves the same goal, and also has the benefit of being fully decentralized [do I need citations?].

We use a graph $G = (V, E_G)$ to represent the communication topology between clients, where $V = \{1,...,K\}$. If client i would receive a message containing the model parameters

from client j, then a corresponding edge $(j, i) \in E_G$ represents this connection. The consensus algorithm updates the model weights using the following expression:

$$W^i \leftarrow \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} (W^j - W^i), \quad \mathcal{N}_i \triangleq \{j : (j, i) \in E_G\}$$

In other words, each client updates its model by adding the average difference from its neighbors in the topology G. This procedure is guaranteed to converge to the centroid of all clients' models, as long as G is fully connected. In addition, the rate of convergence is determined by the second eigenvalue $\lambda_{2,G} \geq 0$ of G's Laplacian L_G. (Add directed graph case)

Notice that the FedAvg algorithm is not easily generalized to the consensus algorithm. In the consensus algorithm, all clients update simultaneously in one iteration. In FedAvg, however, only the central client updates its model in the first iteration, and this client does not even have a model to start with; the other clients get updated in the second iteration, and this update is a replacement, instead of an averaging step.

When using the consensus algorithm, we can observe a tradeoff between having a dense and a sparse network. A denser network would generally have a larger $\lambda_{2,G}$. The rate of convergence is proportional to $e^{\lambda_{2,G}}$ (the distance to the centroid is proportional to $e^{-\lambda_{2,G}}$), so denser network could result in faster convergence. On the other hand, a sparser network could require less communication between clients at each iteration, and could guarantee convergence as long as the graph is connected.

In 2000, Watts and Strogatz revealed the Small-World phenomenon in their seminal paper. The observation is that, starting from a circular graph topology where each node only connects to the few closest nodes ("local" connections), the eigenvalue λ_2 could be drastically increased by randomly rewiring some of those local connections to long-distance pairs. With enough rewiring, we can increase the convergence rate of the consensus algorithm, while not incurring additional communications. This communication scheme allows us to maintain a sparse yet effective setup. In the next section, we attempt to combine the segmented federated learning approach with the small-world topology.

C. Simulation Results of Segmented Federated Learning with Small-World Topology

In this segment, we present the experimental results of combining the two sections above. We tried using layers ($W^j_{L_i}$), output channels ($W^j_{L_{i,out},c_i}$), and kernels ($k^j(i, c_{i-1}, c_i)$) as the segment unit. The kernel-level sharing proved to be taking too long in the simulation, and is unfeasible. The layer-level should be equivalent to the work by [8], so we're left with channel segments.

For the sake of conveniency, assume for now that the communication graph is undirected. Figure 7 compares the effects of sharing with different topologies and with changing segment assignments. In the figure, "Portion of segment units shared" (call it PSS) indicates the percentage of total segments (output channels) that are shared by each client -

if $PSS = 0.5$, for example, then each client j would choose half of its segments $W^j_{L_{i,out},c_i}$. We denote the set of chosen segments' indices as $SegInd^j$, where each item is a tuple $(i, c_i) \in SegInd^j$, i indicates the corresponding layer, and c_i the index of the shared output channel at this layer. Note that segments from all layers i are considered in one pool, instead of being separately picked per layer; in other words,

$$\forall i, j, \ \frac{|\{(i,c) \in SegInd^j\}|}{C_i} \neq PSS = \frac{|SegInd^j|}{\sum_{i=1}^{l} C_i}$$

The random seed was fixed for each experiment.

The figure compares the performances of three different settings.

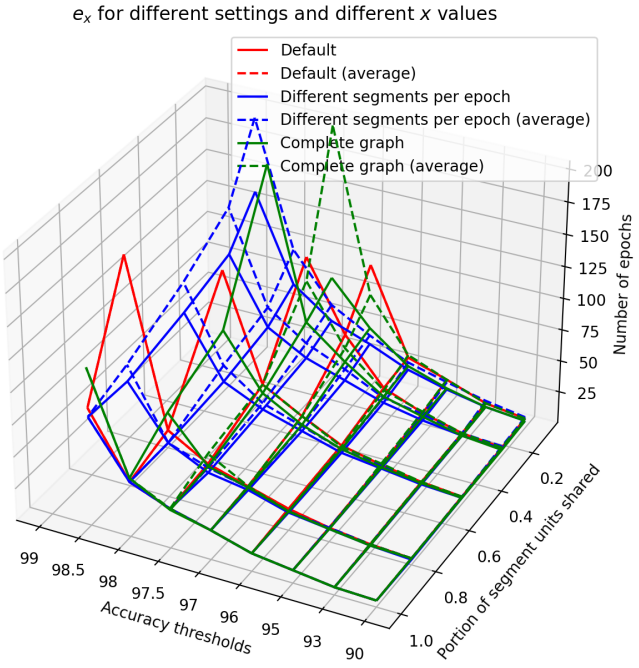
- The first setting, named "Default", pre-determines the segment indices $SegInd^j$ that each client j would poll (receive) from others. Here, we do not necessarily guarantee that $SegInd^{j_1} = SegInd^{j_2}$ for any pair of clients (j_1, j_2). The set of clients that j would poll from is exactly $\mathcal{N}_j$, the set of neighbors of j in the small-world communication topology.

- The second setting, "Different segments per epoch", means that $SegInd^j_t$ for client j is dependent on the epoch number t; at each new epoch, the segment indices is randomly selected again. Everything else is the same as "Default".

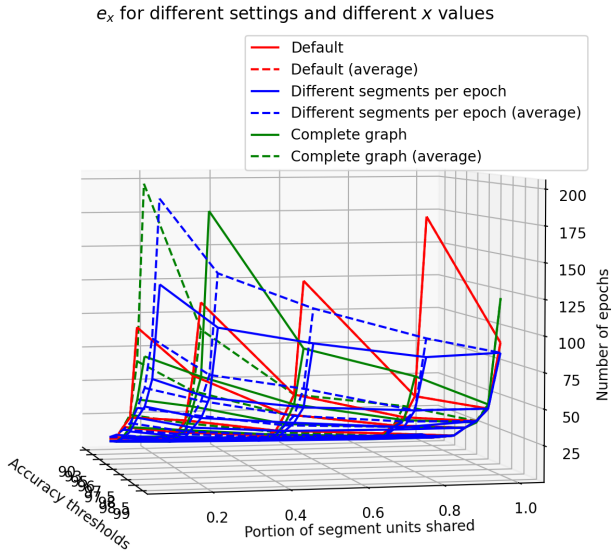
- The third setting, "Complete graph", uses a complete graph as the topology, which means $\mathcal{N}_j = \{k : 1 \leq k \leq K, k \neq j\}$. The others are the same as "Default".

All the experiments were run with the default hyperparameters, such as $E = 0.05$, training length is 200 epochs, training data is partitioned across clients IID with no global or local sharing, etc., and they were run with 5 values of $PSS = \{0.1, 0.25, 0.5, 0.8, 1.0\}$. Figure 7(a-b) display the number of epochs it takes to reach the accuracy thresholds, from e_{90} to $e_{98.5}$ for each PSS value with each setting; the solid lines show the performance of the client with the least training loss at each epoch, measured by the test set accuracy, and the dashed lines show the clients' average performance on the test set.

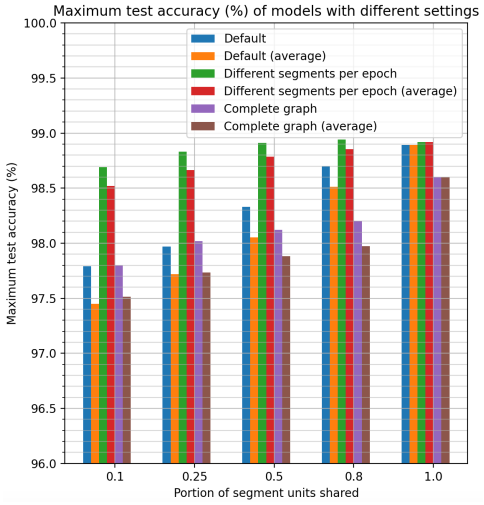
One important thing to note when reading the figures: If an accuracy threshold was never reached, then the corresponding value won't be plotted. For example, the "Default" at $PSS = 0.1$ never reached an accuracy over 97.5%, so the maximum value it has was at $e_{97} = 100$. Thus, even though it appears that the "different segments" setting has the highest values at $PSS = 0.1$, those values represent the accuracies that the other two settings never achieved within 200 epochs. This is validated by Figure 7(c) as well.



(a) Epoch counts



(b) Epoch counts, another angle



(c) Max accuracy

Figure 7: (a-b). Number of epochs to reach e_{90} to $e_{98.5}$, and (c). Max accuracies, for the top-behaving model and the model averages, using different aggregation settings.

With that in mind, the figures give the following observations:

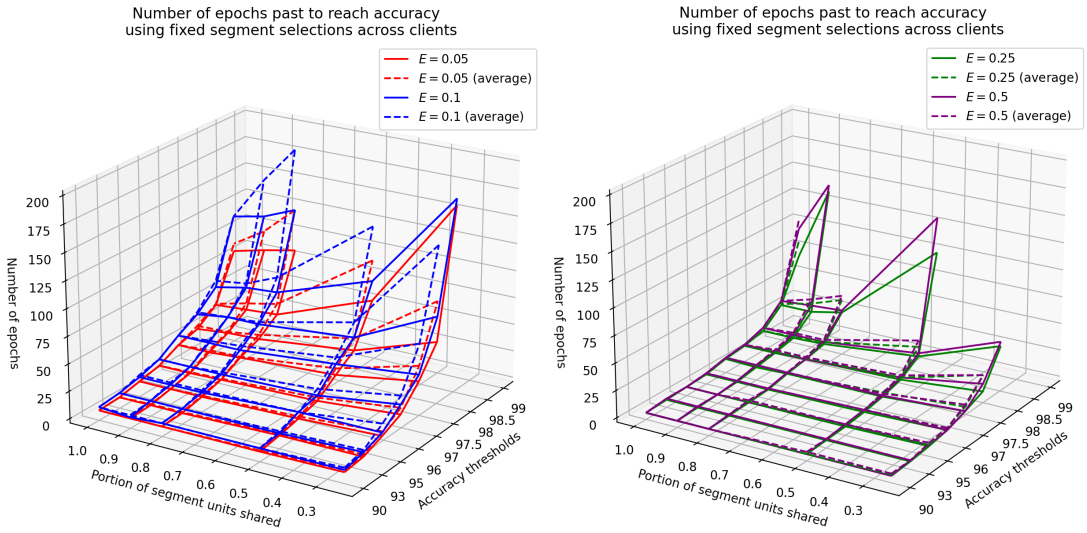
- Larger PSS values helped the models to reach a certain accuracy faster, and made the models able to reach a higher accuracy, for all 3 settings. This is expected, because more aggregation means more weights would be the same across clients at the start of next training epoch.
- Generally speaking, the performance of the "Different segments" setting is better than the "Default" setting, and the ones with complete traph topology had surprisingly worse performances in terms of top accuracy. This observation can possibly be explained by two interpretations:
  - Polling different segments at the end of each epoch was helpful for the model to update all segments with an even chance, thus helping the client train its model parameters close to a set of weights that could fit other clients' data well.
  - Too much aggregation served as a hindrance to better training, and forced the models to perform optimization within a suboptimal region without much exploration.

We also note that the special case when $PSS = 1.0$ should make the learning procedure equivalent to FedAvg algorithm. However, it appears that the three settings ended up at different performances. It is possible that it's a result of different random behaviors (each setting might use different numbers of random actions at each epoch, thus changing the training outcome), or some of the model aggregation steps did not reach full consensus within the given number of iterations. The exact reason is to be determined.

We are also interested in how the update frequency E affects the training in this segmented setting, which is explored in Figure 8 and 9. Both figures display the number of training epochs each model had to go through to reach certain accuracy thresholds in (a-b), the performances by the top-performing clients in (c), and the range of whole model distances between all pairs of clients in (d), and all subfigures display the best-client performances and the average performances. Each color corresponds to a different E value from $0.05, 0.1, 0.25, 0.5, 1.0$. Models with $E = 0.05$ and 0.25 were trained with 200 epochs, while models with $E = 0.1$ and 0.5 with 100 epochs.

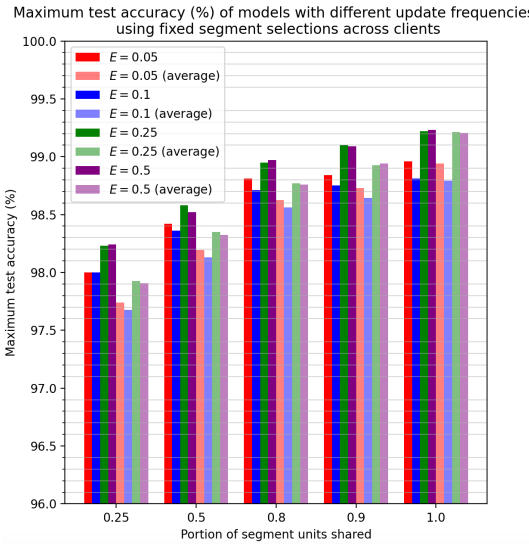
Note that the PSS values tested in those figures are $0.25, 0.5, 0.8, 0.9, 1.0$, where we replaced the previous 0.1 with 0.9 for better observation near the transition point.

Figure 8 tries with different update frequency values E with fixed segment polling for each client: $SegInd^{j_1} = SegInd^{j_2}$, for any (j_1, j_2), and does not change over epochs. In the figure, (a) compares the amount of epochs required for models to reach certain accuracies with $E = 0.05$ and $E = 0.1$. Because the models with $E = 0.1$ has doubled number of samples and optimization steps than models with $E = 0.05$ do, the curves for $E = 0.1$ was multiplied by 2 to better reflect the difference between the two cases. The same was done on curves for $E = 0.5$ in (b) to compare with $E = 0.25$. This was also the reason why $E = 0.1$ was trained with 100 epochs, while $E = 0.05$ had 200 epochs.

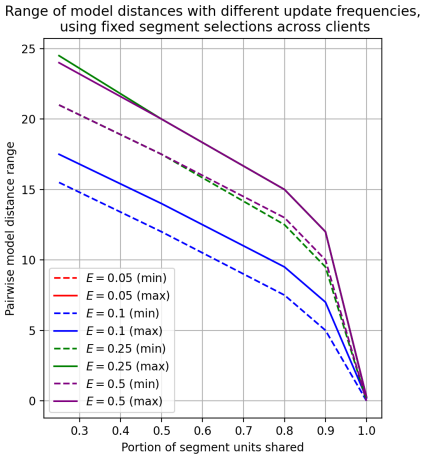


(a) $E = 0.05, 0.1$

(b) $E = 0.25, 0.5$

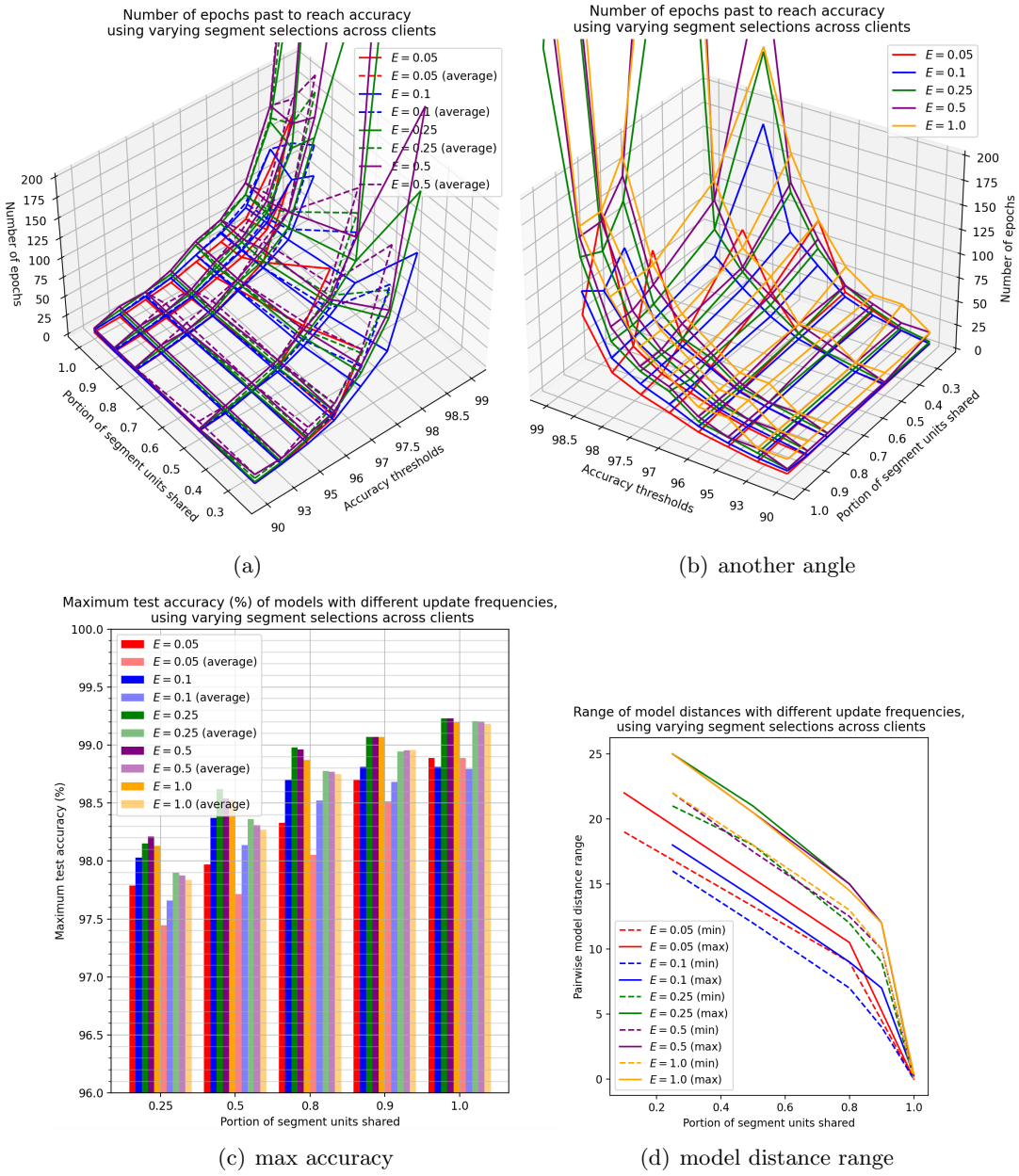


(c) max accuracy



(d) model distance range

Figure 8: (a-b) Number of epochs to reach $e_{90} - e_{98.5}$ for the top-behaving model and the model averages, (c) max accuracies, and (d) range for pairwise model distances in ℓ_2 norms, compared when using different E when clients always share the same set of segments. Values in (a-b) for $E = 0.1$ was multiplied by 2 to compare with $E = 0.05$; the same goes for $E = 0.5$ for $E = 0.25$. Data for $E = 0.05$ in (d) is missing.



(a) (b) another angle

(c) max accuracy (d) model distance range

Figure 9: (a-b) Number of epochs to reach $e_{90} - e_{98.5}$ for the top-behaving model and the model averages, (c) max accuracies, and (d) range for pairwise model distances in ℓ_2 norms, compared when using different E when each client sends different set of segments. Values in (a-b) for $E = 0.1$ was multiplied by 2 to compare with $E = 0.05$; the same goes for $E = 0.5$ for $E = 0.25$.

In both subfigures, the ones with higher E would take a larger number of effective epochs to reach the same accuracy level. In fact, if we put all four curves together, and multiply them with the corresponding constant multipliers $\frac{E}{0.05}$, then we would still observe the same trend (plot not shown due to space constraint, but uploaded as cunoCRS_e_view3.png).

In Figure 9(a-b), different clients poll different sets of segments, similar to the "Default" setting in Figure 7. Additionally, all the curves were placed in the same figure. We observe similar behaviors than in 8; the only difference is that the $E = 0.05$ case turned out to take longer than the others to reach certain accuracies when PSS is low.

Additionally, both figures contain a plot depicting the range of model distances at each epoch: $\min_{j_1,j_2\in[1,K]} \|W^{j_1} - W^{j_2}\|^2$ and $\max_{j_1,j_2\in[1,K]} \|W^{j_1} - W^{j_2}\|^2$. These ranges would be helpful in revealing the underlying relationship between models.

We can make the following observations from Figure 8 and 9:

- It appears that the effective number of training required to reach certain accuracy would be fewer for smaller E values (higher update frequencies). However, this might be an artifact of round-up error, and is not conclusive.
  A more conclusive experiments should use models that learn much slower than the current CNN model, in order to display the zoomed-in behavior. For example, one could experiment with a simpler model structure with fewer expressivity, or use a harder dataset like CIFAR, so that models could not reach an accuracy of 97% after training on only $5 \cdot \frac{E}{K} = 2.5\%$ of the entire dataset.

- Furthermore, as shown in both subfigures labeled with (c), the top accuracy is generally higher with less frequent aggregations (higher E values). This is consistent with the general observation that training is more effective with less frequent parameter changes.

- Similar to experiments in Figure 7, the performance would increase as PSS increases, as shown in (a-c). This is the case regardless of what segments are polled by each client.

- Both subfigures labeled (d) indicate that pairwise model distances never reach 0 until PSS reaches 1, where the entire model is shared between clients under the small-world topology. As expected, the range of model distances monotonically shrinks as PSS increases, because more parameters are shared and synchronized at the start of each epoch. In addition, less frequent aggregations generally resulted in larger model distances.

It is well-expected to see model distances stay positive when each client only poll the same segments from others - the segments that are not shared could achieve similar performances and predictions without having similar weights. It is also not unexpected that model distances still stay positive when each client poll a different set of segments - while it

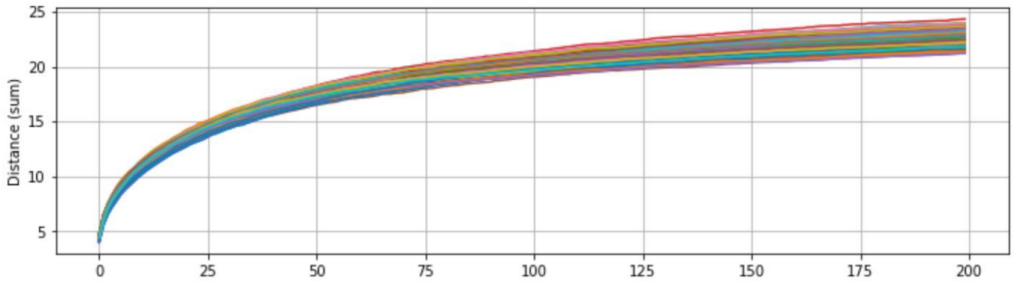


Figure 10: Pairwise model ℓ_2 distance range at each epoch, obtained from the $PSS = 0.25, E = 0.25$ experiment with different segment sets polled by each client.

should be expected that

$$\bigcup_{j=1,\ldots,K} SegInd^j = \bigcup_{i=1,\ldots,l} \{(i, c_i) : 1 \leq c_i \leq C_i\}$$

i.e. the polled segments should cover all existing segments when combining all the clients' $SegInd^j$ sets, it is not guaranteed that each segment is shared across all the clients. If we examine the sharing connection graph G_{i,c_i} for a specific segment $W^j_{L_{i,out,c_i}}$, its edges $E_{G_{i,c_i}}$ would only be a subset of the edge set E_G for the small-world connection graph of the client network G. Its expected size is $\mathbb{E}[|E_{G_{i,c_i}}|] = PSS \cdot |E_G|$, and randomly choosing this many connections from G could easily result in a graph with unconnected components. Figure 10 is a plot showing the pairwise model distance history for such a specific case where $PSS = 0.25$ and $E = 0.25$. The pairwise distances are increasing over time, even though the models were training and aggregating normally. This is observed in all experiments shown in Figure 8 and 9 where $PSS < 1$. Thus, we propose an alternate segment sharing method. Instead of relying on the original Small-world communication topology between clients, we could randomly generate a small-world sharing topology graph G_{i,c_i} for each segment W_{L_i,out,c_i}. At each aggregation iteration, each client j would go over each of its shared segments with index $(i, c_i) \in SegInd^j$, and share the segment with its neighbors in G_{i,c_i}. Figure 11 includes results that compare the previous two settings and this setting, which we labeled as "Different topology per segment" in the legend. The PSS value again determines how many segments would be shared in this way, and those chosen segments with $SegInd^j$ would be the same across all clients.

Additionally, we added a fourth setting, labeled as "Directed, Different segments and topology". In this setting, each segment still follows a small-world connection topology that is different from the other segments' in the aggregation process, However, to simulate the stochastic factors in real application that could affect the communication, this small-world connection topology is randomly formed at each epoch, and each client j would stop polling as long as it has obtained messages containing PSS of the total number of segments. The

following describes this procedure for client j if it were in a real application:

1. $polledSegmentCount \leftarrow [0, ..., 0]$; $segmentGraphLambda \leftarrow [0, ..., 0]$;
2. while True:
3. if segment (i, c_i) from client k is received:
4. $polledSegmentCount[(i, c_i)] += 1$;
5. Update $segmentGraphLambda[(i, c_i)]$ from the received message;
6. If $|\{(i, c_i) : polledSegmentCount[(i, c_i)] > 0$ and $segmentGraphLambda[(i, c_i)] > 0\}| \geq PSS \cdot |W^j|$:
7. break;

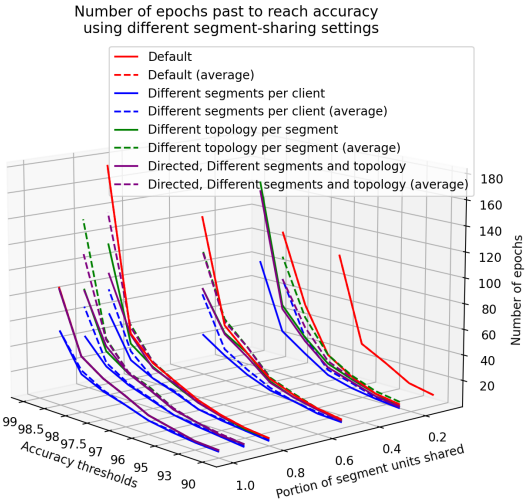
We assume that each message regarding $W^j_{L_i,out,c_i}$ would also contain a list of existing edges for this segment, so that the client could estimate if the communication topology has already become a complete graph. The time it takes for each segment's message to reach client j is fully stochastic, so the set of segments that are aggregated for j, and the set of clients whose segment weights were used during the aggregation for each segment is also fully random.

In simulation, however, we could simplify the procedure as follows, for each epoch in clieng j:

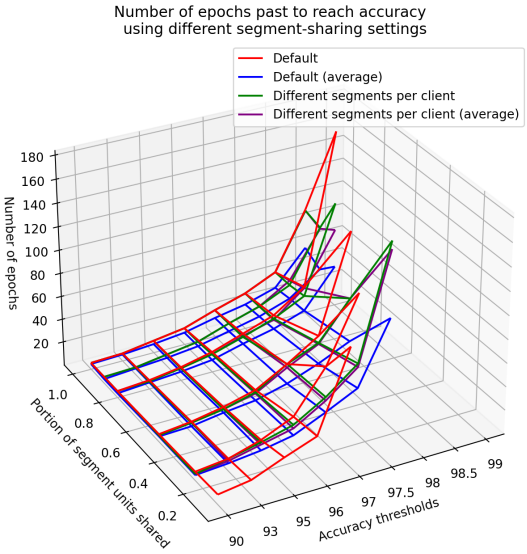
1. Randomly generate $SegInd^j$ with size $PSS \cdot |W^j|$;
2. for (i, c_i) in $SegInd^j$:
3. Randomly generate a small-world topology G_{i,c_i};
4. Poll the segment from all its neighbors, and perform aggregation;

Notice that this procedure does not have guarantee of an undirected update graph for each segment like in the other settings.

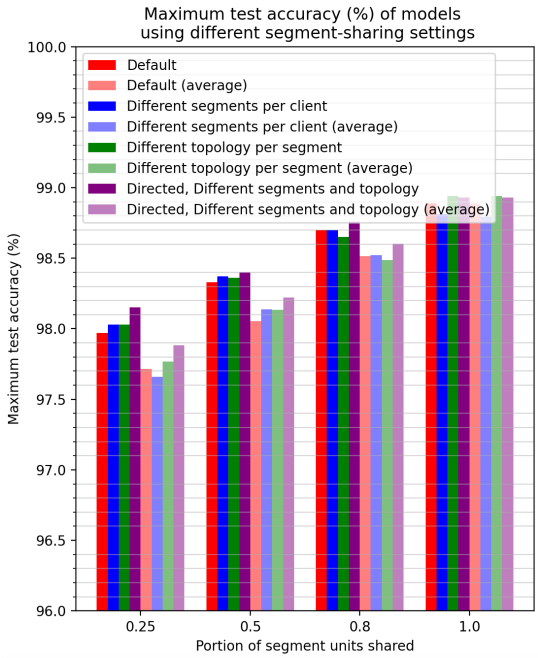
In 11, we compare the performances of all 4 settings. Subfigures (a-c) all used experiments with $E = 0.05$ for 200 epochs of training, but (d) had to use $E = 0.1$ with 100 epochs for the first two settings, because the data of the $E = 0.05$ run was accidentally overwritten.



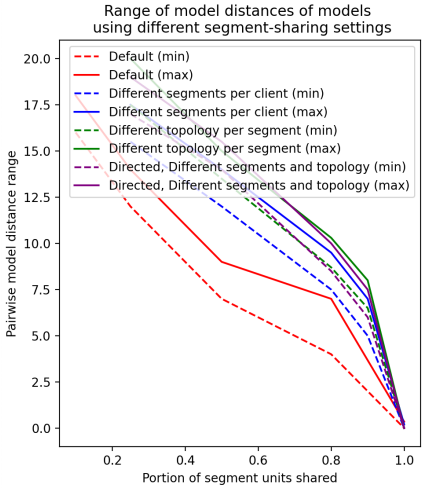
(a)



(b) another angle without averages



(c) max accuracy



(d) model distance range; "Default" and "Different segments" ones used $E = 0.1$ instead of $E = 0.05$

Figure 11: (a-b) Number of epochs to reach $e_{90} - e_{98.5}$ for the top-behaving model and the model averages, (c) max accuracies, and (d) range for pairwise model distances in ℓ_2 norms, compared between different sharing settings described in the legend.

In Figure 11(a-b), the general trend is that "Different segments" setting has the fastest rate to reach certain accuracy thresholds; the settings with segment-wise topology graphs often had similar performances. On the other hand, Figure 11(c) reveals that all settings had very similar performances in terms of peak accuracy, and (d) shows that the model distance history was also similar.

In particular, note from Figure 8(d) and Figure 9(d) have revealed the trend that model distances would reduce with smaller E. Thus, if Figure 11(d) were showing the results with $E = 0.05$ from all settings, then the distance ranges from "Default" and "Different segments" would have been even smaller. This observation has counted against the purpose for the "Different topology" aggregation scheme - while the scheme was designed as an attempt to guarantee convergence for each segment, it overlooked the fact that $PSS < 1$ would mean some segments never get updated this way. It achieves the same thing that the "Default" setting does.

Unfortunately, experiments show that the segmented ideas in this section is possibly still impractical. For example, the simulation time of segmented approaches is significantly longer than the most basic approach, which could be attributed to the way the segment-wise assignments and computations were implemented. By treating each segment separately, the code fails to utilize the array layout efficiency that modern machine learning packages could achieve.

We have also documented the communication costs of the procedure, represented simply as the combined sizes of shared segments at each epoch for now. The results are shown in 12, where the y-axis unit is in millions of floats. The figure compares the 4 settings studied in Figure 8-11 (with at most 20 iterations in the model aggregation step, to reduce computation time), and also includes result from settings shown in Figure 7 (with at most 100 iterations).

As expected, larger PSS values results in larger communication costs, but more factors come into play. Between the first four settings, the amount of segments and the amount of edges in communication graphs should roughly add up to the same at a fixed PSS, so the differentiating factor should be the amount of iterations required to reach consensus.

- The settings using fixed topology ended up with 3 times less communication cost than the ones using segment-wise topology. In real applications, the segment-wise small-world option would require more communication to verify the connectivity of the graph, so the communication costs would be even larger. This idea is thus impractical.

- It is possible that some implementation error existed in the experiments. For example, when $PSS = 1$, all 4 settings should be effectively doing the same procedure - the entire model should be shared with others, and the only difference is whether the others receive the whole model or segments of the model. The difference between results with and without segment-wise topology indicates possible errors.
  One possible explanation is that the "different topology" settings did not actually

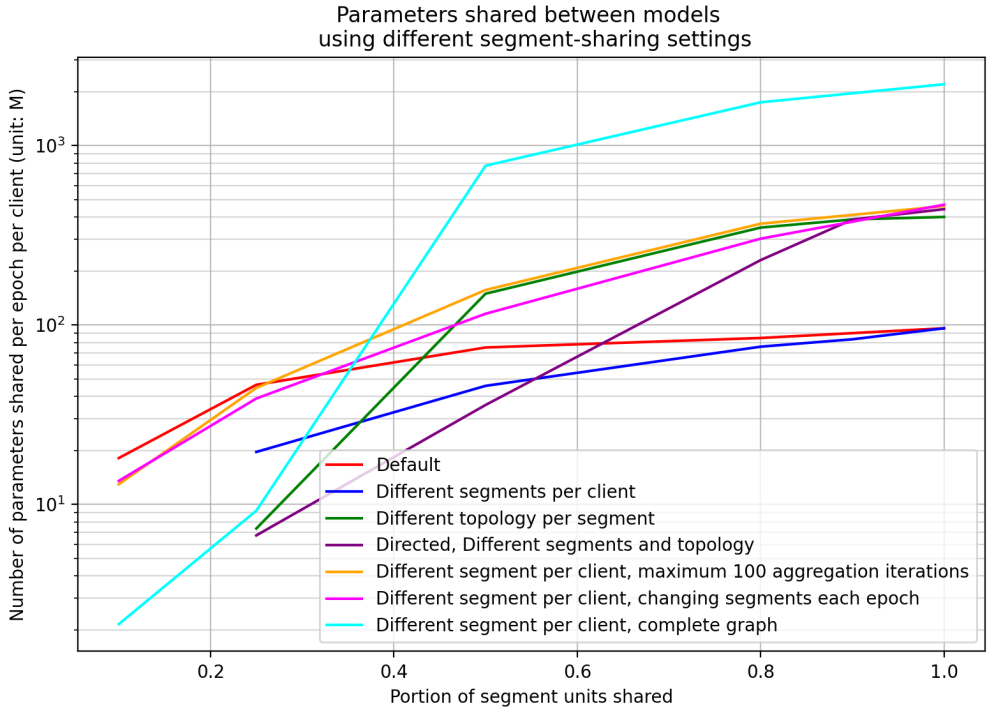


Figure 12: Average number of parameters shared by each client during each epoch's aggregation, compared between different sharing settings described in the legend. The last 3 labels had 100 as the maximum number of averaging iterations, while the others had 20.

manage to converge to average model segments after 20 iterations for all segments. Thus, the models would keep exchanging weights until the maximum iteration is reached. This explains the difference of 3 times, where the ones with fixed topology could converge in about 6 iterations using the small-world topology. On the other hand, the settings with different topologies managed to converge faster with $PSS = 0.25$. One suspected reason is that the topology-generating code had a rare chance of generating one that is not fully connected, which caused all but a few of the convergence to fail.

One other possible explanation is that the linear weights in a convolution layer were not shared as components, but their differences were still considered when evaluating the segment disagreement values against a certain error threshold to determine if consensus has been reached. If this is the case, then the implementation is subject to further debugging.

- The result between "Different segments" with 20 and 100 maximum iterations is also hard to interpret. They have similar shapes, and the difference is roughly 5 times,

equal to 100/20, possibly indicating that the models never reached convergence? The source is probably also implementation error?

- The complete graph line demonstrates the trade-off of having more connections well. As mentioned in previous parts, denser connection means faster convergence, as well as more messages per iteration. The amount of messages required for the complete graph could be written as:

$$PSS \cdot |W| \cdot K(K-1) \cdot \frac{\alpha(PSS|W|)}{e^K} \cdot segment\_size$$

where e^K represents the rate of error decrease where K is the second eigenvalue of the Laplacian for a complete graph, $segment\_size$ is the average number of floats each segment contains, and α is a variable describing other factors, such as number of iterations.
On the other hand, the amount of messages for the Default setting could be:

$$PSS \cdot |W| \cdot K \cdot \frac{\alpha_{SW}(PSS|W|)}{e^{K/\beta}} \cdot segment\_size$$

where we use β to reflect the decrease of convergence rate.
Thus, theoretically, the complete graph should either always have less cost, or have more cost, depending on whether the ratio $\frac{K}{e^{K-K/\beta}}$ is less or greater than 1. However, it appears that α/α_{SW} is not a constant in the experiment, and has rather been a nonlinear term. It is also possible that the rounding of $1/e^K$ affected the values.

Finally, several experiments were run using channels as segments, but purely with directed graphs instead of undirected ones. All the resulting models performed poorly.

D. Future Work

First and foremost, works in the future should check if the observations described above would still hold true for a task that is harder to train on. For example, using a simpler model with fewer parameters to learn MNIST classification, or using the same model in this paper for CIFAR-10 classification, would both prolong the learning process, and allow us to "zoom in" the graphs over epochs further. In addition, more complex benchmark tests would allow for less frequent model aggregations - as we have observed, less frequent aggregation would yield better and faster performances, but the current model cannot afford to do so if the accuracy reaches 99% within the first epoch.

In addition, it is important to derive and analyze theoretical results of the methods above, if feasible. By finding theoretical bounds of the training procedure, or finding equivalencies between experiments listed above, the testing could become less complicated.

The segmentation procedure requires optimizing the current implementation, as well. The smaller the segments, the more of them needs to be shared, and this results in larger

overhead costs. Instead of reducing segment sizes, perhaps a better way is to reduce the resolution of the shared parameters, so that each client's shared parameters could be represented with much less bits - for example, Reisizadeh et al. proposed a variation algorithm, FedPAQ, that aimed at reducing communication cost by reducing the update frequency and quantizing the update process [12].

Alternatively, the communication cost could be reduced by limiting the parameter structures. For example, if we approach the image classification task not with a CNN model, but with a graphical model such as a Markov Random Field, then the number of shared parameters could be reduced to the graph topology and the edge potentials. Past work regarding consensus with Bayesian Networks also exist [13]. With the rise of interest in Graphical Neural Networks (GNN), there is much potential for this idea.

Possible future steps include sharing simplified models with similar performances, using ideas of transfer learning in decentralized ways, training Spars Convolutional Neural Networks (SCNN, developed in 2015 by Liu et al. [14]) to reduce cost, etc. Some researchers have also discovered that models could grow further apart (measured by mutual information) while still increasing their parameters' correlations, thus showing us a different explanation to why the whole model distances were growing with each epoch. [15]

3.2.2 Different Optimization Methods

The idea of ensemble learning is using different kinds of models on the same dataset, and then creating a master model that combines the predictions of each trained model, and try to yield an optimized combination method to reduce error [citation needed]. We could borrow this idea and train several models simultaneously, with the same structure but different training parameters. Then, during the aggregation, the averaging of model parameters can help pull these models' weights together. If federated learning is robust against such averaging, then this could serve as a quick way of tuning models and finding the better set of training parameters for a given task.

Below we investigate several different training parameters that could be altered, and see if Federated Learning with the naive averaging aggregation approach is robust with such methods.

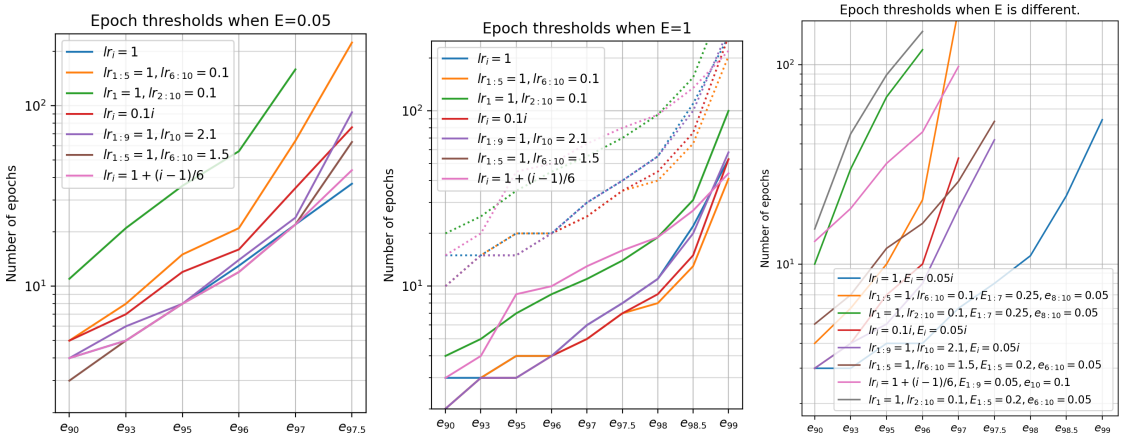
A. Different Learning Rates

Below we use Adadelta as all the clients' optimizer, with learning rate ranging from 0.001,0.01,0.1,1, and 5 between clients. We chose the hyperparameters to be $K = 10$; in addition, to stabilize the training, we implement training data sharing with $S = 0.1$ and $\sigma^2 = 0.7$, two parameters introduced from the previous sections. Let lr_i be the learning rate of client i. We tried different combinations of learning rates, as well as different combinations of learning speeds. The resulting training performances can be seen in Figure 13. We tried different combinations of learning rates: 1) having half the clients with a slower rate, and the other half with a faster rate; 2) giving one client a learning rate different from

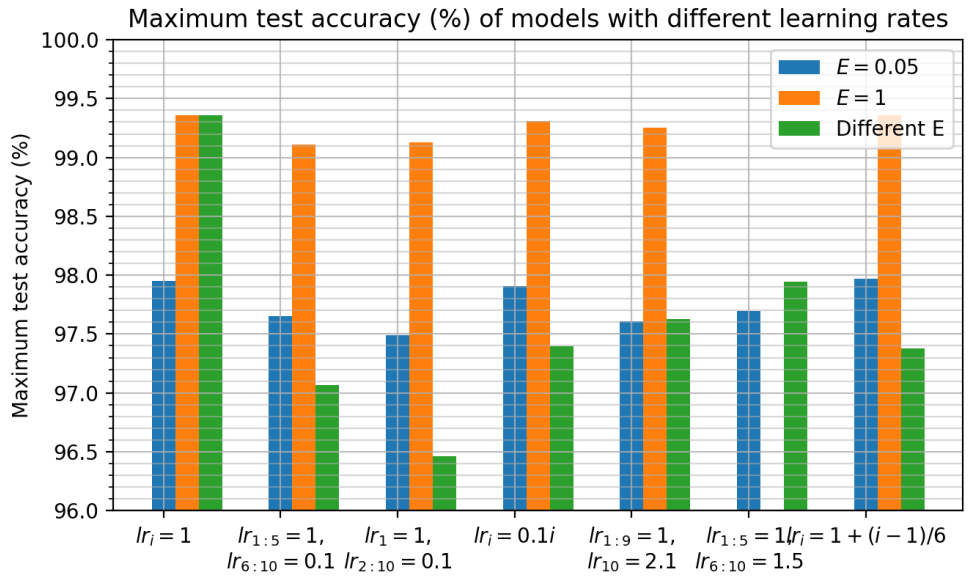
the others; 3) keeping a continuum of learning rates across the clients. The results, tested under $E = 0.05$ and other default settings, are plotted in Figure 13(a). In addition, Figure 13(b) shows the performance when $E = 1$, with dashed lines multiplying the solid lines by 5 to try to match the trend in (a) to no avail. Those figures show that the clients could train successfully in this setting, but no meaningful conclusions could be drawn here. Even the order by which each set of clients reach a certain accuracy is different in both figures.

Figure 13(c) further complicates the matter by introducing different learning speeds. If agent i has learning speed $E_i = 5E = 0.25$ while others have speed $E = 0.05$, then client i's model gets updated 5 times more than other clients within one epoch by seeing 5 times more samples. We combined different learning speeds with learning rate distributions, and all the models were able to converge.

The results show that, if the learning rates are conservative, then varying learning rate did not significantly affect the performance. In addition, for MNIST, the models perform better if update frequency is kept at $E = 1$ instead of $E = 0.05$.



(a) $E = 0.05$ (b) $E = 1$ (c) Various E



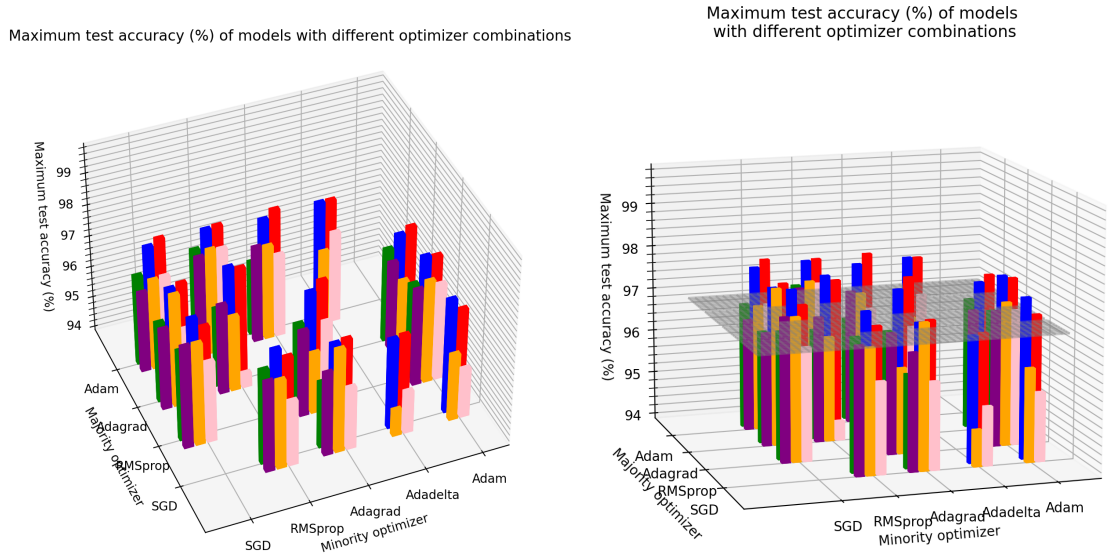
(d) Maximum test accuracy percentages

Figure 13: (a-c): Epoch threshold values with different combinations of learning rates and learning speeds over 500 epochs, as specified in the legend. (d) Maximum test accuracy values of models in (a-c); the missing bar is where models diverged.

B. Different optimization algorithms

We mix-and-match different default optimizer options offered by PyTorch that would suit for the image classification task. Namely, there are five common choices, Adam, Adagrad, Adadelta, RMSprop, and SGD.The learning rate for each optimizer is chosen empirically: 0.001, 0.01, 0.005, 0.0005, and 0.05, respectively. It is worth noting that each optimizer exhibited similar behavior compared to part A when having different learning rates among clients. We test 3 different combinations: 9 optimizers of one type plus 1 optimizer of another type; 7 of one type plus 3 of another; and 5 of each type.

Again, we chose the training parameters to be $K = 10, E = 0.05$; in addition, we use $S = 0.1$ and $\sigma^2 = 0.7$, two parameters introduced from the previous sections, to stabilize the training. Figure 14 plots the maximum accuracy achieved by each of the combination that was tested.

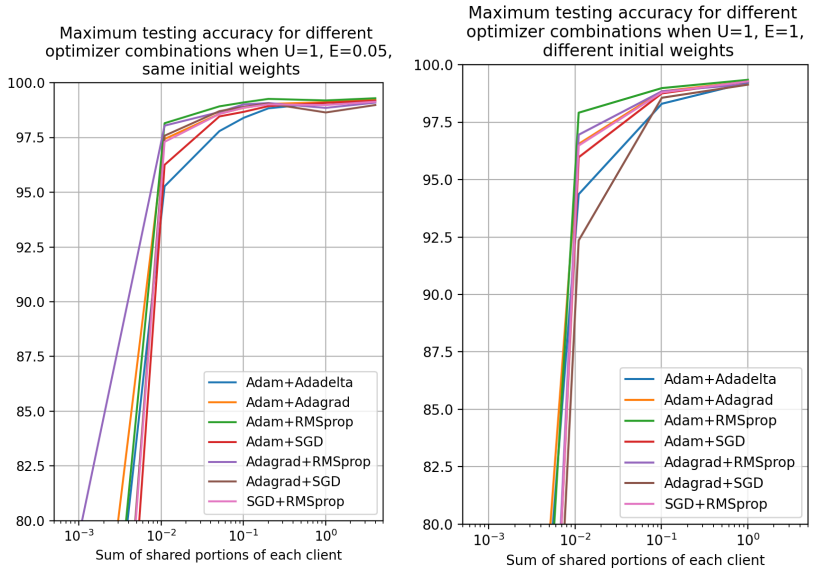


(a) (b) With the plane marking 97%

Figure 14: Maximum test accuracy for clients with mixed optimizers, trained over 200 epochs; empty positions represent combinations that were not run.

The results reveal that combining any two optimizing methods could lead to convergence to a satisfactory model even without uniform initial weights, for the small-scale MNIST dataset with only $K = 10$ clients. However, the resulting accuracies are slightly lower than other models, which is made clear by the low number of pairs that managed to exceed an accuracy of 97% (shown as the gray grid in the middle). It is worth noting that, combining clients 3 or even 6 different optimizers, still resulted in a convergence to models with similar performances.

In addition, the models with different optimizers and skewed training data were still able to converge to a satisfactory global model, as long as it has a small percentage of globally shared data, as shown in Figure 15. In other words, changing optimizers did not change established behavior.



(a) $E = 0.05$ (b) $E = 1$

Figure 15: Maximum test accuracy for clients with mixed optimizers, trained over 200 epochs, with fully skewed datasets.

C. Different Loss Functions (0212-0226)

Two loss functions that are pertinent to image classification problems are included in the PyTorch library: The binary cross entropy loss, and the . While loss functions reflect different goals of training, we speculate that some of the goals can be generalized. As a proof-of-concept, we identify two possible loss functions for image classification problems - Negative Log Likelihood Loss and Binary Cross-Entropy Loss - both available in PyTorch library. We then combine the two loss functions to $K = 10$ clients in different proportions, and perform standard FedAvg algorithm with default parameters. The result is as follows:

The problem is that those experiments weren't well-documented. Mostly it just records observations. The observation is that the loss function didn't matter too much.

3.2.3 Local training after global aggregation (unfinished idea; only temptive results)

Results show that continuing training on local data after clients have achieved a well-performing global model would hurt the performance.

3.2.4 Preventing local minimum by adding noise to local models (unfinished idea; only temptive results)

Adding certain noise to model weights after the global model has fell into a local minimum. We started off training on the MNIST dataset with different initial weights for clients, and the models performed poorly after 50 epochs. Next, we inject a Gaussian noise (parameters=?) to each client's model weights: $W_i^p(t+1) = W_i^p(t) + \eta_i$ where p indicates an arbitrary parameter in the model. When federated learning resumed, 3 of the clients were able to acquire a model with accuracy $> 97\%$, while the rest were still stuck in the local minimum. (Needs more experiment to see what's going on)

4 Future Ideas

If we want it to work on a directed graph, then we might have to use training schemes like push-sum, as mentioned by [Nedić et al. 2018] to guarantee convergence.

One other idea is to have each client solve one particular classification problem (i.e. hold 10 clients, and each client i is responsible for training a set of parameters w_i that predicts the probaility that the given image belongs to class i), and see if this still fulfills the decentralized optimization problem definition (is the objective function decomposable in this way?). One may also consider using the ADMM procedure, as outlined by [Boyd et al. 2010] in section 7.

Another topic of interest is checking the effect of periodically applying random noises to some of the models, as a way to "nudge" models off from possible local minima. With multiple clients sharing their parameters in the federated learning framework, it could be possible that this model would encourage some of the clients to find a successful alternate direction. This idea could also be implemented by giving different momentum values for different clients, where clients with higher momemtum values during optimization would tend to explore more, while clients with lower momentum would hold them back at each update step to prevent from possible divergence.

Federated Learning could also apply in other settings. For example, consider Federated Reinforcement Learning - naturally, it would suit a multi-agent setup. Agents could communicate each other's findings through shared parameters, and converge to certain behavioral models. One other way to adapt this is to set it as a parallel training procedure, similar to ensemble learning, where multiple agents learn in the same environment with different parameters and hyperparameters, and we try to locate one good model from them.

5 Acknowledgement

This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA.

References

[1] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[2] Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. Peer-to-peer federated learning on graphs. *arXiv preprint arXiv:1901.11173*, 2019.

[3] Stefano Savazzi, Monica Nicoli, and Vittorio Rampa. Federated learning with cooperating devices: A consensus approach for massive iot networks. *IEEE Internet of Things Journal*, 7(5):4641–4654, 2020.

[4] Akihito Taya, Takayuki Nishio, Masahiro Morikura, and Koji Yamamoto. Decentralized and model-free federated learning: Consensus-based distillation in function space. *arXiv preprint arXiv:2104.00352*, 2021.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Basic mnist example. `https://github.com/pytorch/examples/tree/master/mnist`. Accessed: 2021-01-30.

[7] PACE. *Partnership for an Advanced Computing Environment (PACE)*, 2017.

[8] Chenghao Hu, Jingyan Jiang, and Zhi Wang. Decentralized federated learning: a segmented gossip approach. *arXiv preprint arXiv:1908.07782*, 2019.

[9] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data, 2018.

[10] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. SCAFFOLD: Stochastic controlled averaging for federated learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5132–5143. PMLR, 13–18 Jul 2020.

[11] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. Federated learning with matched averaging, 2020.

[12] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031. PMLR, 2020.

[13] Jose M Pena. Finding consensus bayesian network structures. *Journal of Artificial Intelligence Research*, 42:661–687, 2011.

[14] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.

[15] Peng Xiao, Samuel Cheng, Vladimir Stankovic, and Dejan Vukobratovic. Averaging is probably not the optimum way of aggregating parameters in federated learning. *Entropy*, 22(3):314, 2020.