Hyperparameter Tuning in a Federated Learning Context

Xander Pero
Department of Industrial Engineering
University of Illinois at Urbana-Champaign
Champaign, IL, USA
apero2@illinois.edu

Abstract

There has been great progress in creating algorithms designed for federated learning, but the methods for tuning the hyperparameters of these models has lagged behind. Traditional hyperparameter optimization methods are not directly applicable to the context of federated learning because the data is split amongst many clients. Optimal hyperparameters on one client are unlikely to remain optimal for other clients in the case of non-iid data. The computation cost of hyperparameter optimization may be too great for some applications of federated learning, namely when connecting to mobile devices. In this work, I examine the practical applications of hyperparameter tuning with the Nelder-Mead algorithm, a derivative-free optimization heuristic. I show that the behavior of Nelder-Mead for an algorithm is influenced by its own design, and therefore is more or less suitable for certain types of problems. I also show that it is possible to alleviate a too-small initial learning rate. The model including hyperparameter tuning converged faster in the convex case and reached a higher peak accuracy after training.

1. Introduction

Federated learning has been a hot topic in machine learning since its proposal by (McMahan et al., 2017). The concept of distributed optimization has been explored before to various degrees, but federated learning proposes its own unique challenges. In both scenarios, there is a central server and a number of external clients, each containing its own dataset. Moreover, all of the data is unable to be stored to the same, central location, meaning that one cannot run an optimization algorithm on the whole dataset at once. Instead, the algorithm must run on the local datasets, extract model parameters after training, then aggregate and incorporate the results into the global model. This establishes some amount of data privacy, which is becoming increasingly important as governments consider the ramifications of using consumer data. Some laws such as HIPAA in the United States and GDPR in the European Union are already in place.

Federated learning imposes its own set of requirements on top of the distributed optimization framework. First, not all clients are available for training in a given communication round. Considering that one main application of federated learning is with mobile data, devices will only be available for training during non-use and when charging. Second, the communication between the server and an individual client is unreliable and slow. To that end, traditional distributed optimization algorithms that communicate back and forth between the server and clients are infeasible for the federated learning setting. Third, the data on each client is not independently and identically distributed amongst the clients. All clients draw from the same unknown, global distribution, but each client may have a different representation of the data. Fourth, the data is massively distributed and unbalanced. There are a very large number of clients relative to the number of datapoints on a given client, and the amount of data varies amongst clients.

Federated learning is concerned with minimizing the following problem: $min\, f(w) = min\left(\sum_{k}^{K} f_i(w)\right)$

The function $f_i(w)$ is the per-client loss, which can be rewritten as $L\left(x_i, y_i; w\right)$.. Because the relevant datasets are concerned with classification, cross entropy loss is used for each local loss function. As noted by (McMahan et al., 2017). communication costs outweigh computation costs for local devices. Therefore, reducing the number of bits communicated, and by extension the number of communication rounds between the client and server, is the goal for a federated learning algorithm. Several methods of reducing bits transferred are outlined by (Konečný et al., 2016). These methods can be applied to existing federated learning algorithms and hyperparameter tuning in practice to reduce communication costs.

The FedAvg algorithm proposed in (McMahan et al., 2017) was the first to reduce communication costs by increasing the quality of a single communication round of local updates. The global loss function is approximated by the sum of all local loss functions; because each client is assumed to draw from the global distribution, this assumption is valid. By increasing the number of gradient steps on a given device, the global model converges faster than a federated version of SGD under some conditions. However, FedAvg in general can struggle with non-iid data (Zhao et al., 2018) particularly because of the difference in local and global loss minimization. Local convergence does not imply global convergence, which is exacerbated by the effect of non-iid data.

Recent algorithms have been built on top of this framework and in response to data heterogeneity. FedProx adds a regularizing term to the local loss function which induces a penalty relative to the norm of the model weight difference between the global and local model (Li et al., 2020). SCAFFOLD corrects the overfitting towards the local model with a control variate, resulting in faster convergence than FedAvg in the non-iid scenario (Karimireddy et al., 2020). Lastly, FedDyn uses a dynamic regularizer within the local loss function such that the local and global stationary points are aligned at the limit, which allows for exact minimization at the local level (Acar et al., 2021).

One benefit of automatic tuning is resilience against poor initial hyperparameters. For the federated setting in practice, it would be costly to do a trial run to optimize hyperparameters by examining the results after a certain number of communication rounds. An existing algorithm, FLoRA performs single-shot hyperparameter optimization in that it performs hyperparameter optimization locally, then minimizes loss for an aggregated loss surface and a global validation set (Zhou et al., 2022). A second benefit of hyperparameter tuning would be a lower loss reached after training because of a decreasing step size. For constant step size, SGD converges in distribution near a stationary point, but does not converge to the stationary point exactly. Heuristics such as learning rate decay per round or successive halving (SHA) exist and could be applied to federated learning. Smartly dropping the step size by analyzing when convergence has slowed has been explored (Lang et al., 2019) but it is unclear how to apply this in the federated setting. This could be included in the FedAvg algorithm or its variants, but considering that the statistical test would run on the individual local datasets, the learning rate drop would fire when a local dataset is nearing true convergence, which is contradictory to inexact minimization, which is needed for non-iid data. Moreover, the local number of epochs may not be enough to collect the required information to drop the learning with significant evidence.

In this work, I attempt to improve the baseline federated learning algorithms by applying an existing hyperparameter tuning method, namely the Nelder-Mead algorithm. Invoking hyperparameter optimization at the start of training can result in quicker initial convergence, though it is also possible to run Nelder-Mead after a certain number of iterations. Assuming that loss from SGD has converged in distribution near a stationary point, dropping the step size is required to decrease the loss further. I experiment with different values of the Nelder-Mead algorithm, namely the number of local epochs within the hyperparameter optimization loss function and maximum iterations of Nelder-Mead comparisons.

Related Work. The FedAvg algorithm has been shown to be faster than SGD in iid and somewhat non-iid settings (McMahan et al., 2017). However, the effects of non-iid data have drastic effects on the convergence speed of FedAvg, as examined by (Zhao et al., 2018). Recent algorithms tackle this problem via a regularizing term in FedProx (Li et al., 2020), control variates (Karimireddy et al., 2020), or a dynamic regularizer which aligns the global and local stationary points at the limit (Acar et al., 2021). Notably, there is a difference between inexact and exact minimization, as pointed out in (Acar et al., 2021). FedAvg, FedProx, and SCAFFOLD do not have local convergence implying global convergence, which means that inexact minimization must be used to avoid overfitting to the local model in a heterogeneous scenario. It has been shown in (Karimireddy et al., 2020) that varying the number of local epochs affects the convergence speed of these algorithms. Specifically, with a greater number of local epochs, SCAFFOLD and FedAvg converge slower for non-iid data, but converge faster for even 10% similar data. The practical ramifications of this are that statistical heterogeneity of the clients affects the optimal hyperparameters of the local model.

Hyperparameter tuning has been applied to federated learning recently. It is possible to estimate hyperparameters at the beginning of training, as shown by Zhou et al., 2022). There, they use a single round of communication to determine sufficiently optimal hyperparameters based on a global validation set. On the other hand, it is possible to incorporate hyperparameter tuning within the algorithm itself. (Khodak et al., 2021) use local validation data to evaluate the hyperparameter optimization results, which are then used to update the hyperparameters of the global model at the same point of updating the model weights. In order to overcome computational costs of hyperparameter optimization, they employ weight-sharing, a commonly used technique in neural architecture search. Aside from the FedEx model itself, they propose a successive halving algorithm for the federated setting, which can reduce to random

```
Algorithm 1: Nelder-Mead Method

Initialize: Construct simplex $\left\{a_0^1, a_0^2, \ldots, a_0^{d+1}\right\}$
**For** $e = 0, 1, \ldots, E$ **do**
    **For** $k = 1, 2, \ldots, d + 1$ **do**
        Calculate $l^k = L\left(a^k\right)$
    **End for**
    Order $l^k$ and let $m = argmax\left(l^k\right)$
    Calculate $c = \sum_{i \neq m} a^i$
    Reflect;
    Expand;
    Contract;
    Shrink-Contract;
**End for**
```

search with a single elimination round and choosing the best hyperparameter combination from a single run. These two works are different from this in that FLoRA attains hyperparameters that will be in place for the rest of training. This approach leverages the first benefit of tuning: better starting hyperparameters and increased convergence speed. On the other hand, FedEx and SHA make use of the first benefit, but more so the second where an adaptive step size may improve model performance at convergence.

The previous works of FedDyn, SCAFFOLD, and FedProx are concerned with comparing a novel algorithm against FedAvg and the other algorithms in a homogeneous setting. For example, FedDyn has the capability to use exact minimization, but for fairness of comparison, they used SGD with a larger number of epochs relative to the other algorithms. In this work, I focus on evaluating an algorithm against itself with and without Nelder-Mead hyperparameter optimization. Project success will be determined by the effect of Nelder-Mead iterations to adjust learning rate on convergence speed.

1. Nelder-Mead Method

The Nelder-Mead method is a derivative-free optimization heuristic. This algorithm constructs a simplex with $d + 1$ vertices, where d is the dimension of hyperparameter space.in the hyperparameter space then evaluates the function value at each point in the simplex. Once each point is evaluated, the function value is ranked, and the centroid of all points except the worst is computed. The algorithm then tries a new test point, then adjusts the simplex based on the function value of the newly tried point relative to the function of current vertices of the simplex. This is repeated until convergence or stopping iteration. Here, the function being evaluated is the local cross entropy loss after a specified number of SGD iterations. With the constraints of federated learning, Nelder-Mead must be run on each local dataset independently.

The Nelder-Mead algorithm requires a conjoined training and testing sequence within a loss-returning function that it minimizes. This function creates a deep copy of the original model such that gradient updates can be performed with the trial learning rate without affecting the current model. Once these updates have reached the max iteration of local epochs, a separate number of epochs specifically for the Nelder-Mead step, the model is then evaluated using a local validation set.

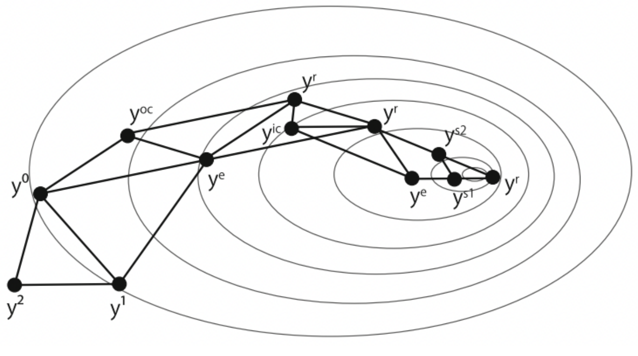


Figure 1. Sample Progression of Nelder-Mead algorithm

The Nelder-Mead algorithm is described in Algorithm 1. The loss function $L\left(a^k\right)$ is the cross entropy loss of a single client after a specific number of rounds of SGD and maximum iteration of Nelder-Mead iterations, E. Reflect, Expand, Contract, and Shrink-Contract are auxiliary functions which determine the location of the replacement point of m, an example learning rate in hyperparameter space.

Let a_d be the rank-d point, meaning second worst only to a_m. Also let a_1 be the rank-1 point, meaning it is the best point. The optimal action can be calculated by relating $L(a_d)$ and $L(a_1)$ to a newly calculated guess.

1. Reflection. New point $a_r = c + \eta(c - a_m)$. If $L(a_u) < L(a_r) \leq L(a_v)\ \exists u, v \neq m$, $\eta = 1$, replace a_m with a_r. Therefore, a_r is not the worst nor best point on the newly constructed simplex including a_r instead of a_m. The new point is reflected across the old centroid and will have a better value than the previous iterate.
2. Expand. New point $a_e = c + \gamma(a_r - c)$. If $L(a_k) \leq L(a_r) \forall k$, $\gamma = 2$, move further along this direction by replacing a_m with a_e instead. With this update, the simplex becomes larger and moves further away from the replaced point than Reflection, with likely better vertices.
3. Contract. If $L(a_r) < L(a_k)$ then the reflected point is worse than the original worst vertex. Then create new point $a_c = c + \beta(a_m - c)$, $\beta = 0.5$. New simplex has better vertices than the original.
4. Shrink-Contract. If $L(a_c) < L(a_m)$ then the contracted point is worse than the previous worst point. Redefine all non-best points such that the jth point $a_j = a_1 + \delta(a_j - a_1) \forall j$, $\delta = 0.5$. This operation is computationally expensive, though rare, and may be replaced with a simple Contract.

Implementation. For a communication round with hyperparameter optimization, the server sends the current model weights to selected clients. Then, each client runs the Nelder-Mead algorithm on its local dataset which returns a learning rate. Then, the local device performs the regularly-scheduled gradient updates per the update algorithm: FedAvg, SCAFFOLD, etc. which are then transmitted back to the server along with the client's new learning rate. This learning rate is likely to be different from the resulting learning rate from another client. The server adjusts model weights and any additional variables according to the update algorithm. Finally, the server averages all new learning rates and sets the new learning rate for all clients equal to that learning rate. This new learning rate is only determined by the resulting rate of these clients that performed Nelder-Mead; the previous learning rate of unselected clients do not contribute to the new learning rate.

```
Algorithm 2:
Learning Rate Averaging

Input: T, $P_k$, $E_h$, $M_h$
**For** $e = 0, 1, \ldots, E$ **do**
  **For** client $k \in P_k$ **do**
    **If** $e \bmod T = 0$
      $\theta_k$ = Nelder-Mead($E_h$, $M_h$)
      $w_k$ = ClientUpdate()
  **End for**
  $w$ = GlobalUpdate()
  **If** $e \bmod T = 0$ **do**
    $\theta = \frac{1}{|P_k|}\sum_k \theta_k$
**End for**
```

This algorithm is summarized in Algorithm 2. S_h is the number of communication rounds that pass before the next round of Nelder-Mead is run. For example, if $S_h = 50$, then hyperparameter optimization is performed in communication rounds 0, 50, and so on. It's important to note that the transmission of learning rate results in a small communication cost. However, this cost is negligible compared to the number of weights within the neural network and infrequent communication of learning rate. P_k represents the set of selected clients for the given communication round. Nelder-Mead() corresponds to the Nelder-Mead algorithm which outputs the new learning rate, and ClientUpdate() is the usual local update for the update algorithm of choice with the new learning rate. GlobalUpdate() is the aggregation of all the local updates in the communication round. Finally, the new learning rate is calculated by averaging the learning rates of selected clients.

The other two hyperparameters introduced with the Nelder-Mead algorithm are the max number of epochs within the SGD loop within Nelder-Mead, denoted E_h, and the maximum number of Nelder-Mead iterations, denoted M_h. These two variables greatly impact the behavior of Nelder-Mead. E_h can be interpreted as search depth for the local problem. If E_h is small, then the algorithm will adjust the learning rate to where it can make the most progress in that many steps, therefore continually increasing the learning rate. On the other hand, if E_h is large, then the algorithm will look for step sizes that have the best loss after that many steps, and provide a more stable result. To that end, There is a tradeoff between computation power and solution stability, though there are practical ways to alleviate this, such as upper bounding the learning rate and starting with a cautiously small learning rate.

The other hyperparameter, the number of Nelder-Mead iterations, denoted M_h, affects how much the learning rate can change in a round of Nelder-Mead. Consider $M_h = 1$. In this scenario, the learning rate is only allowed to make a single update per client per communication round, which can help prevent overfitting to the local model. This furthers the inexact optimization paradigm; the global model is required to make P training iterations before the learning rate can be adjusted once again. On the opposite end, M_h can be large, which would result in Nelder-Mead convergence. The biggest concern here is that with a small E_h, for example $E_h = 1$, and a sufficiently large M_h, then the Nelder-Mead algorithm reduces to line search such that the local model would converge within a single step if the learning rate was unbounded.

My implementation of FedAvg, FedProx, SCAFFOLD, and FedDyn was taken from the official codebase of (Acar et al., 2021). On top of the already existing model comparison, I added functionality for learning rate scheduling selection, whether it remained constant, decayed, or used Nelder-Mead. This included the loss-returning Nelder-Mead algorithms for each update algorithm and storage of hyperparameters over time into the already existing data storage capabilities. The train-test split from scikit-learn is used to split local data into training and validation. Nelder-Mead is implemented using scipy.

I applied this method to the existing FedAvg and SCAFFOLD algorithms, though there is also the possibility to use Nelder-Mead with FedProx and FedDyn. Because FedDyn uses exact minimization, there is no step size to tune. However, the codebase from (Acar et al., 2021) implements FedDyn using SGD for fairness amongst algorithms, and as such the code has the current capabilities for FedDyn, though the results may not be particularly relevant.

FedProx and FedDyn have an additional regularization hyperparameter which can be tuned. However, because any positive value for the regularization term would increase the local loss, the Nelder-Mead algorithm would quickly reduce it to zero or near zero, removing its effect and worsening the overall global performance. Therefore, running Nelder-Mead or another hyperparameter optimization algorithm that utilizes local loss should also avoid adjusting these regularization terms. A future work could include a separate way of tuning the regularization term or by adapting the loss function that Nelder-Mead minimizes so that the optimal regularization term for minimized local loss is not always zero.

2. Experiments

Experiments were run on a subset of the EMNIST dataset, EMNIST-L, which only contains 10 labels (Cohen et al. 2017), and a synthetic dataset. Other datasets were not used because of the intensive time to train and evaluate the model. Of the available datasets, EMNIST had the least-complicated neural network, resulting in significantly lesser training time than the other datasets. The EMNIST model is the same architecture as in (Acar et al., 2021). It contains two hidden layers and 100 neurons. For the EMNIST dataset, both iid and non-iid splits are tested, with each client receiving 480 datapoints. The non-iid splits are created using a Dirichlet distribution. Parameter settings of 0.3 and 0.6 are used in accordance with (Acar et al., 2021). The iid split is created by randomly sampling the datapoints.

The synthetic dataset was generated using the same process as (Acar et al., 2021), though there are 40 clients rather than 20 which were originally used. $\left(x_j, y_j\right)$ pairs are generated using $y_j = argmax\left(\theta_i^* x_j + b_i^*\right)$, where $x_j \in \mathfrak{R}^{30\times1}$, $y_j \in \{1, 2, 3, 4, 5\}$, $\theta_i^* \in \mathfrak{R}^{5\times30}$, and $b_i^* \in \mathfrak{R}^{5\times1}$. The optimal set of parameters for device i is $\left(\theta_i^*, b_i^*\right)$. Each element is drawn from $N\left(\mu_i, 1\right)$ where $\mu_i \sim N\left(0, \gamma_1\right)$. Datapoint features are modeled as $\left(x_j \sim N\left(\upsilon_i, \sigma\right)\right)$ where σ is a diagonal covariance matrix with elements $\sigma_{k,k} = k^{-1.2}$. Each element of ν_i is drawn from $N\left(\beta_i, 1\right)$ where $\beta_i \sim N\left(0, \gamma_2\right)$. The original FedDyn paper considers a varying number of datapoints per client which utilizes another parameter γ_3, though that is not utilized here. $\left(\gamma_1, \gamma_2, \gamma_3\right)$ correspond to (α, β, γ) in the code implementation. Each client has 200 datapoints, and values for iid, $\alpha = \beta = \{0, 0.5, 1\}$ are tested.

For the EMNIST model, 100 clients, a batch size of 50, a weight decay of 10^{-4}, 10% client sampling, 10 local epochs of the update algorithm, and learning rate of 0.1 are applied for all models. For the synthetic model, 40 clients, a weight decay of 10^{-5}, a batch size of 10, 10% client sampling, and 10 local update algorithm epochs are used for all models.

Gradient Update–Tuning Tradeoff. There exists a tradeoff between performing Nelder-Mead iterations and updating the model gradients. If there is a total budget of the number of SGD steps able to be performed, how should it be split amongst Nelder-Mead and the actual update algorithm? This can be viewed as an exploration-exploitation tradeoff, and previous work (Zhou et al., 2022; Khodak et al., 2021) have quantified the regret of non-optimal hyperparameters. Future work could further explore this work formally as a reinforcement learning problem.

It's possible to establish a budget of SGD steps for all models. This budget is calculated as $\frac{TEn_k}{n}$, where T represents the number of total communication rounds, E represents the number of local epochs, n_k

represents the number of selected clients, and n the total number of clients. In words, the budget is the exact number of SGD steps that a constant or decaying learning rate algorithm would take during training. For an algorithm that includes Nelder-Mead, some of these SGD steps must be allocated towards Nelder-Mead iterations. For each iteration that includes Nelder-Mead tuning, an upper bound on the additional SGD steps is $E_h M_h n_k$, the multiplication of Nelder-Mead epochs, iterations, and participating clients. This is not exact because Nelder-Mead may exit successfully before the max iterations are reached. Therefore, an upper bound on the total number of SGD steps as a function of communication round t for Nelder-Mead can be written as $f(t) = En_k t + E_h M_h \lceil \frac{t+1}{S} \rceil$ where S_h = the number of communication rounds before the next round of Nelder-Mead.

The implementation of Nelder-Mead is then concerned with matching or surpassing the performance of a baseline algorithm from a potentially non-optimal learning rate. If performance is matched, then the model was trained successfully using fewer communication rounds, which returns back to the original goal of reducing communication costs.

Nelder-Mead Testing Set. One point of interest was in determining how the testing set during a Nelder-Mead iteration affects convergence speed. The two options are to use the full training set for Nelder-Mead iterations, then evaluate loss on the same dataset. This utilizes an optimization approach of quickest convergence rather than the machine learning paradigm of splitting training and validation data. The other approach, which was used for all other methods, involved splitting the local dataset into mutually exclusive training and validation datasets. The set of Nelder-Mead hyperparameters were $E_h = 5$, $M_h = 5$, with standard $S_h = 40$, $T = 200$ for EMNIST Dirichlet(0.3).

3. Results

Effect of additional computation within Nelder-Mead algorithm. The effect of additional computation within the Nelder-Mead step is compared on the EMNIST dataset in Figure 2. The tag “nm” indicates that Nelder-Mead was applied, whereas “decay” means that a per-round learning rate decay was applied. Learning rate is displayed to show the influence of learning rate tuning on the algorithm.

One communication round is the full loop of distributing the model, performing the local update, then aggregating the results to the global model. The loss is on the global scale. As seen in Figure 2, Nelder-Mead complexity does not have a significant effect on the convergence speed. There is somewhat decreased loss in the heavy Nelder-Mead model with FedAvg, but an unnoticeable difference in SCAFFOLD. Full details of the model are available in appendix A.1. Also, results on the EMNIST dataset with same model construction, but on an iid split of the data, are also presenting in the appendix A.2.

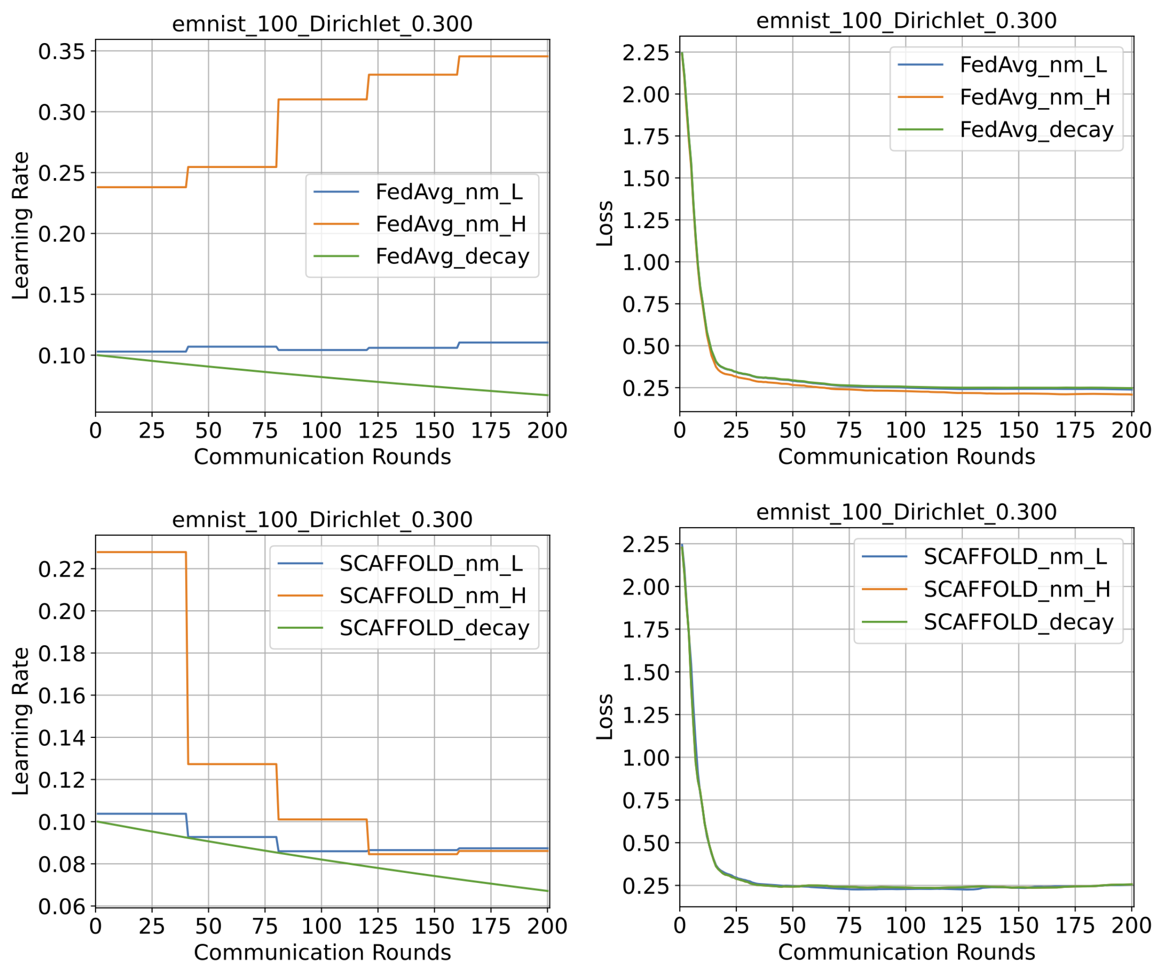


Figure 2. Learning rate and global loss vs. communication rounds for FedAvg and Scaffold with a Dirichlet(0.3) allocation of data. It can be seen that the choice of Nelder-Mead complexity, denoted by L and H as light and heavy complexity, does not make a drastic difference on the loss convergence despite a vastly different learning rate amongst the three runs.

Influence of Nelder-Mead testing set. The results using the full training data set vs. a held-off local validation dataset during learning rate tuning are highlighted in Figure 3. The model was fit with $E_h = 5$, $M_h = 5$, so the Nelder-Mead algorithm is of shallow depth. The learning rate was tuned every 40 communication rounds and the model was trained for 400 communication rounds.The learning rate continues to increase for the model that uses the full training dataset in both parts, whereas the model with split datasets tends to decrease on average. This is likely in response to overfitting to the training data.

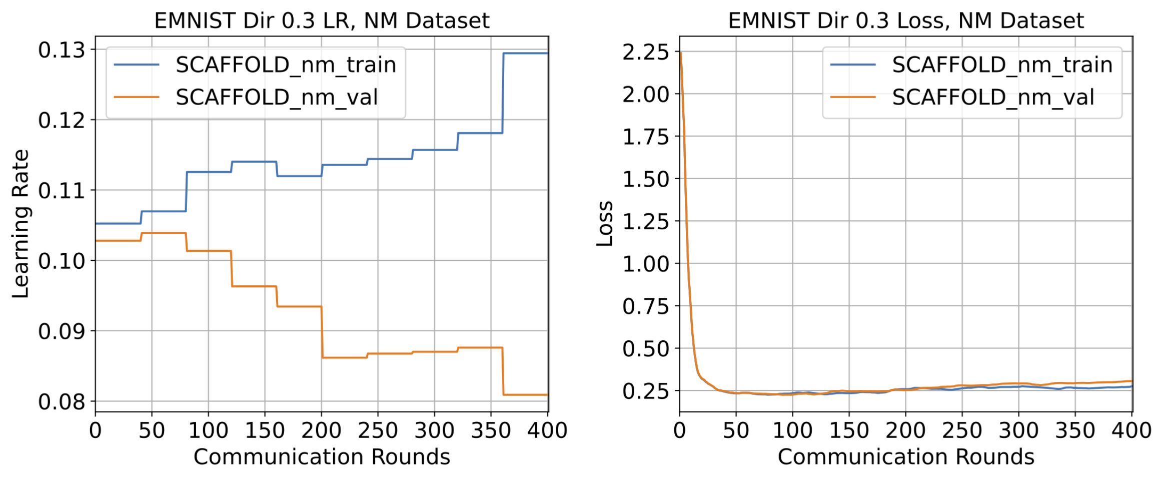


Figure 3. Learning rate and loss vs. communication rounds for EMNIST. The "train" model both runs Nelder-Mead on the entire local dataset then evaluates the loss on the entire local dataset. The "val" model splits the data into a 75-25 train-test split, tunes the learning rate using the training dataset, then evaluates the loss after E_h local epochs using the validation dataset.

Interestingly, there is lesser global loss on the model with the full training dataset used in both parts. This could be due to the increased size of the data, which may outweigh the mutually exclusive split. Moreover, because the data is not iid amongst the clients, the validation dataset is not representative of the global dataset. In that sense, the training dataset is a better approximation of the global dataset, which explains the increasing loss after approximately 200 communication rounds for the split dataset model. Therefore, using the training dataset for both components of hyperparameter tuning may be worthwhile with data heterogeneity.

FedAvg and SCAFFOLD comparison. All models began with a learning rate of 0.1, the usual 200 communication rounds, and learning rate adjustment every 40 communication rounds. It is agreed upon by Figure 4 and Figure 1 that FedAvg has an increased tendency to increase the learning rate relative to SCAFFOLD. In the upper-left graph, the decaying SCAFFOLD and FedAvg models have the same decay rate, so those plots overlap one another. SCAFFOLD enjoys quicker initial convergence, but perhaps due to the model structure itself, has increased loss after it becomes minimized after communication round 80. The bottom right graph shows a zoomed in version that focuses on later convergence after the initial dramatic drop in global loss. The increased loss is present in both the decaying and Nelder-Mead strategies of learning rate adaptation, so this is not an effect of Nelder-Mead tuning. The FedAvg models may have a similar effect after 200 iterations. The increase in loss does not have an effect on test accuracy, which is plotted against the communication rounds in the bottom right figure. Like the loss, accuracy quickly approaches a threshold and then stalls in progress for the remaining number of rounds.

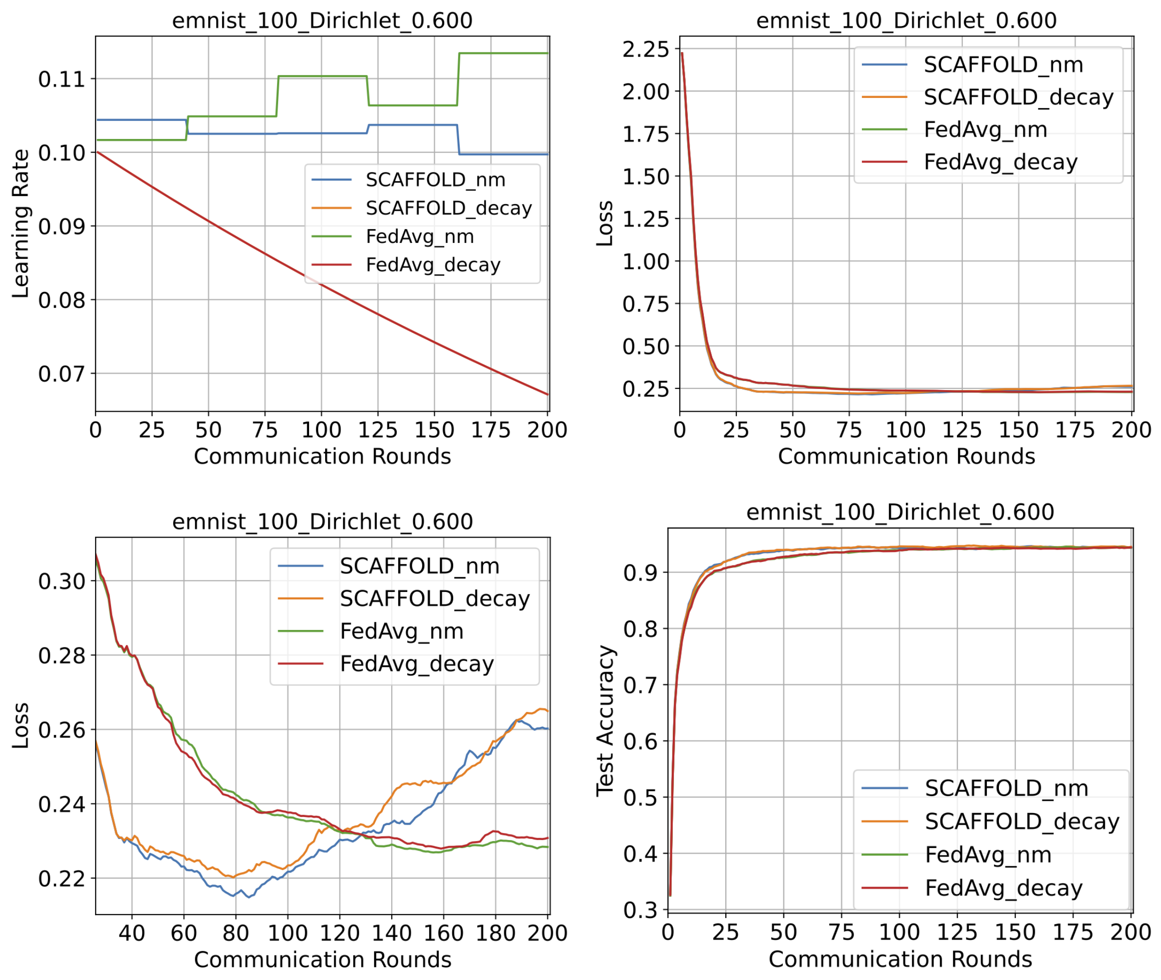
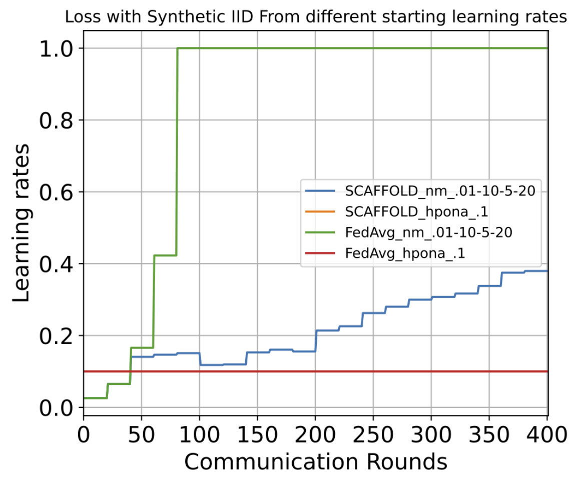
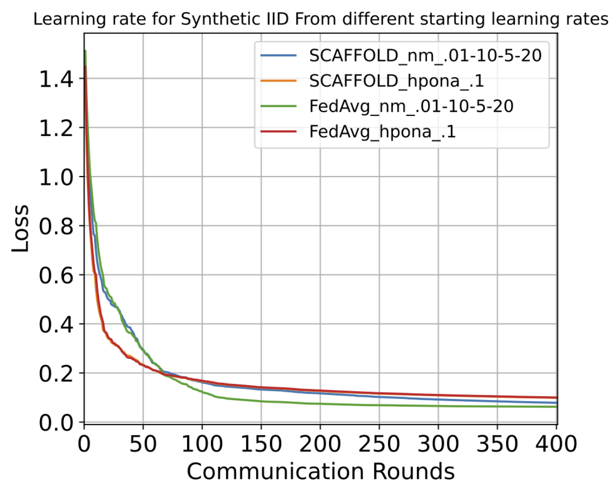
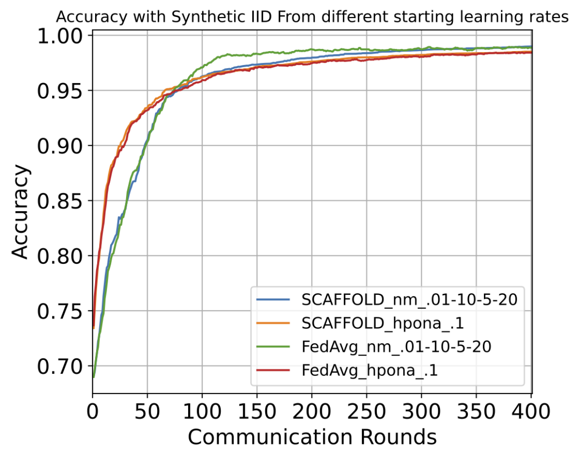


Figure 4. Results of comparing SCAFFOLD and FedAvg with Nelder-Mead and decaying learning rate schedules on the EMNIST dataset with Dirichlet(0.6) dataset split. Learning rate does not change drastically with each Nelder-Mead update because $E_h = M_h = 5$; not much progress can be made in that many iterations and little depth.

Nelder-Mead with non-optimal learning rate. It's shown in Figure 5 that despite a poor starting learning rate, the model including learning rate tuning can converge faster than a constant learning rate model once given the opportunity to adjust its learning rate. On the synthetic data, both SCAFFOLD and FedAvg were run with constant learning rate 0.1. The other model was FedAvg and SCAFFOLD but with tuning, initial learning rate 0.01, trial learning rate epochs $E_h = 10$, max Nelder-Mead iterations $M_h = 5$, and hyperparameter tuning every 20 communication rounds of 400.







Model	Final Accuracy
SCAFFOLD NM	0.989875
FedAvg NM	0.98875
SCAFFOLD Constant	0.985125
FedAvg Constant	0.98425

Figure 5 and Table 1. Results of mismatching initial learning rate on synthetic iid data for SCAFFOLD and FedAvg with $E_h = 10$, $M_h = 5$, and $S_h = 20$. FedAvg with Nelder-Mead reaches near-stationary convergence the fastest and ends with similar top accuracy.

The constant learning rate algorithms show initial increased convergence speed, but they are overtaken by FedAvg after a few updates are made to the learning rate. By iteration 80, the fourth learning rate update, FedAvg has already capped out at the maximum learning rate of 1. On the other hand, SCAFFOLD is not as quick to change its learning rate, and has slower convergence despite the highest final accuracy after 400 communication rounds. The control variates, while able to add additional stationarity to the model, slow down its adaptability, therefore slowing convergence.

4. Conclusion and Future Work

Hyperparameter optimization within the federated learning setting poses unique challenges but can be done with some success. For when additional local epochs may overfit the training model, hyperparameter tuning is a way to make use of additional computational capabilities when it would otherwise be disadvantageous. With automatic tuning, a suboptimally small learning rate can be chosen, then adjusted as training progresses for quick convergence. Finally, the implementation of automatic learning rate tuning can result in improved convergence at the limit when decreasing compared to a constant learning rate algorithm of previous federated learning methods.

Future work may involve formally examining the tradeoff between computation spent on learning rate tuning and gradient updates with a finite budget under multiple problem types. Another form of hyperparameter tuning could occur every communication round with fewer iterations, meaning a small change in learning rate with each communication round. This would incur an additional communication cost of the learning rate per round, but may make up for it with quicker convergence than this proposed method of updating after a certain number of communication rounds.

References

Acar, D. A. E., Zhao, Y., Navarro, R. M., Mattina, M., Whatmough, P. N., & Saligrama, V. (2021). Federated learning based on dynamic regularization. *arXiv preprint arXiv:2111.04263*.

Cohen, G., Afshar, S., Tapson, J., & Van Schaik, A. (2017, May). EMNIST: Extending MNIST to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)* (pp. 2921-2926). IEEE.

Karimireddy, S. P., Kale, S., Mohri, M., Reddi, S., Stich, S., & Suresh, A. T. (2020, November). Scaffold: Stochastic controlled averaging for federated learning. In *International Conference on Machine Learning* (pp. 5132-5143). PMLR.

Khodak, M., Tu, R., Li, T., Li, L., Balcan, M. F. F., Smith, V., & Talwalkar, A. (2021). Federated hyperparameter tuning: Challenges, baselines, and connections to weight-sharing. *Advances in Neural Information Processing Systems*, *34*.

Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., & Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

Lang, H., Xiao, L., & Zhang, P. (2019). Using statistics to automate stochastic optimization. *Advances in Neural Information Processing Systems*, *32*.

Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A., & Smith, V. (2020). Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems*, *2*, 429-450.

McMahan, B., Moore, E., Ramage, D., Hampson, S., & y Arcas, B. A. (2017, April). Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics* (pp. 1273-1282). PMLR.

Ozaki, Y., Yano, M., & Onishi, M. (2017). Effective hyperparameter optimization using Nelder-Mead method in deep learning. *IPSJ Transactions on Computer Vision and Applications*, *9*(1), 1-12.

Zhao, Y., Li, M., Lai, L., Suda, N., Civin, D., & Chandra, V. (2018). Federated learning with non-iid data. *arXiv preprint arXiv:1806.00582*.

Zhou, Y., Ram, P., Salonidis, T., Baracaldo, N., Samulowitz, H., & Ludwig, H. (2022). Single-shot Hyper-parameter Optimization for Federated Learning: A General Algorithm & Analysis. *arXiv preprint arXiv:2202.08338*.

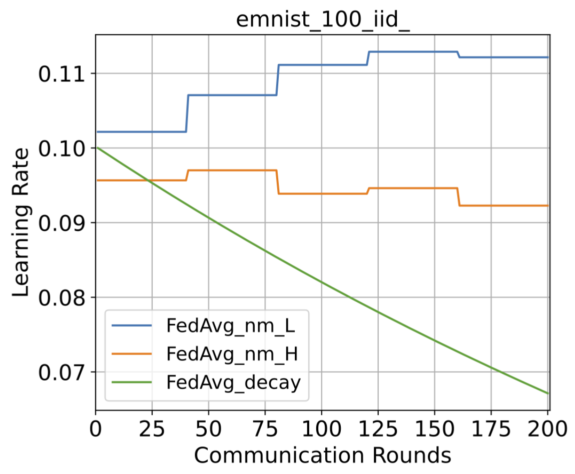
A. Appendix

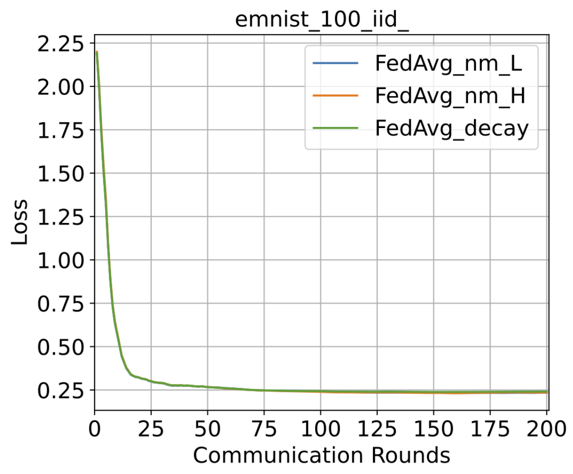
A.1 EMNIST Models L, H, Decay details

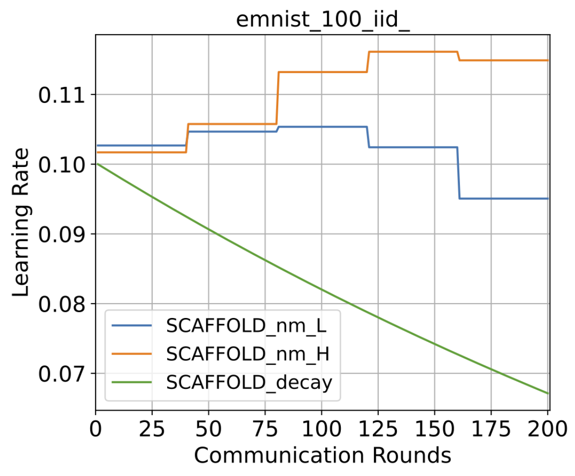
Three sets of parameters were tested on the EMNIST dataset. In all scenarios, there were 100 clients, a weight decay of 10^{-4}, batch size of 50, 10% client participation, 10 local epochs, and Nelder-Mead every 40 communication rounds if at all. Two Nelder-Mead models labeled L and H were used. Model L, the light model, had $E_h = 10$, $M_h = 5$. Model H, the heavy model, had $E_h = 100$, $M_h = 20$. The decay model induced a learning rate decay of 0.998 per round.

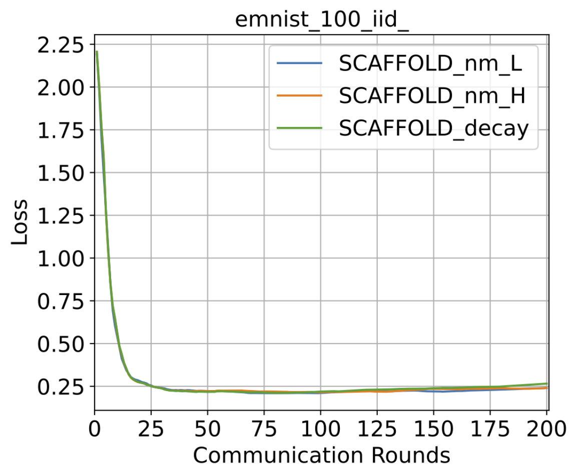
A.2 EMNIST Models L, H, Decay for iid split

Below graphs show the results for the iid split of the EMNIST dataset using the same models as in A.1. There is no significant difference between the models in FedAvg with iid data.


emnist_100_iid_
Learning Rate
Communication Rounds
FedAvg_nm_L
FedAvg_nm_H
FedAvg_decay


emnist_100_iid_
Loss
Communication Rounds
FedAvg_nm_L
FedAvg_nm_H
FedAvg_decay


emnist_100_iid_
Learning Rate
Communication Rounds
SCAFFOLD_nm_L
SCAFFOLD_nm_H
SCAFFOLD_decay


emnist_100_iid_
Loss
Communication Rounds
SCAFFOLD_nm_L
SCAFFOLD_nm_H
SCAFFOLD_decay