Federated Learning with SGD Variants: Implementation, Benchmarking, and Analysis

*

Himanshu Singh *2022217*
Jaleel Ahmed Radhu Khwaja *2022225*

I. Motivation

Federated Learning (FL) has emerged as a compelling paradigm for training machine learning models in a decentralized manner, where data remains on user devices to preserve privacy and reduce communication overhead. Instead of aggregating raw data on a central server, FL relies on iterative model updates shared between clients and a coordinating server. This setting introduces several challenges, including data heterogeneity, limited client participation, communication constraints, and instability in optimization dynamics.

Most federated learning systems are built upon Federated Averaging (FedAvg), which uses standard Stochastic Gradient Descent (SGD) on clients and weighted model averaging at the server. While effective, FedAvg can converge slowly or unstably, especially under non-IID data distributions — a common characteristic in realistic federated settings. This raises an important question:

> *Can more advanced SGD variants improve convergence, stability, or accuracy when adapted to the federated setting?*

Project Goal

The goal of this project is to systematically study how different variants of SGD, when applied either on clients or at the server, impact the efficiency and performance of federated learning. We explore whether adaptive, momentum-based, or accumulator-based gradient methods can offer advantages over FedAvg in realistic FL conditions.

Contributions

Our main contributions are as follows:

- We implement five federated learning algorithms based on well-known SGD variants: FedAvg, FedAdam, FedAdagrad, FedRMSProp, FedAdamW and FedAMSGrad.
- We benchmark each method against the baseline FedAvg algorithm, which uses client-side SGD and weighted server averaging.
- We evaluate performance on the FEMNIST dataset in a non-IID federated setting, using Dirichlet-based data partitioning.
- We compare convergence speed, final accuracy, and training stability across different optimizers.

This study highlights the role of optimizer choice in federated settings and provides insights into when adaptive server-side algorithms offer meaningful benefits over standard FedAvg.

II. DATASET

- The primary dataset used in this project is FEMNIST, a federated version of the original EMNIST handwritten character dataset. It is partitioned by writer, making it naturally suitable for federated learning experiments with non-IID data.
- The dataset consists of 62 classes (digits, uppercase, and lowercase letters), with each client representing a unique writer. FEMNIST contains 3,550 users, 805,263 training samples, and 89,924 testing samples. Each image is a 28x28 grayscale character.
- To simulate varying levels of client heterogeneity, we additionally apply **Dirichlet-based non-IID partitioning** with two values of α:
  - $\alpha = 0.5$ — higher skew, representing stronger non-IID distribution
  - $\alpha = 1.0$ — more balanced, closer to IID behavior

  This allows us to evaluate how different optimizer-based federated algorithms converge under different levels of statistical heterogeneity.

III. PROPOSED ARCHITECTURE

a) Model Architecture.: For all experiments we use a small convolutional neural network (CNN) adapted from the official PyTorch MNIST example and tailored to the FEMNIST setting. Each input image is a $1 \times 28 \times 28$ grayscale character. The network consists of two convolutional layers followed by two fully connected layers:

- Conv1: 32 filters, kernel size 3×3, stride 1, input shape $1 \times 28 \times 28$.
- Conv2: 64 filters, kernel size 3×3, stride 1.
- Max-pooling: 2×2 on the output of Conv2, reducing the spatial resolution from 24×24 to 12×12.
- Dropout($p = 0.25$) after the pooling layer.
- Fully connected layer FC1: $9216\,(= 64 \times 12 \times 12)$ inputs $\rightarrow 128$ hidden units, followed by ReLU.

- Dropout($p = 0.5$) after FC1.
- Output layer FC2: $128 \rightarrow 62$ units, corresponding to the 62 FEMNIST classes (digits, lowercase, and uppercase letters).

The network outputs unnormalized logits; a cross-entropy loss is used during training. This architecture is intentionally lightweight to make large-scale federated experiments feasible.

b) Federated System Architecture.: We adopt a standard star-shaped federated learning architecture with a single central server and K clients. A generic `Node` base class stores the model, device, and learning rate, and is extended by specialized `Client` and `Server` classes for each algorithm (FedAvg, FedAdam, FedAdam_V2, FedAdagrad, FedRMSProp). At the beginning of each communication round, the server broadcasts the current global model to a sampled subset of clients. Each selected client instantiates its own local copy of the CNN, performs local training using its private FEMNIST partition for a fixed number of local epochs, and returns the updated model parameters and local dataset size to the server.

The server then aggregates these updates using the algorithm-specific rule: FedAvg performs weighted averaging, while FedAdam_V2, FedAdagrad, and FedRMSProp apply Adam, Adagrad, or RMSProp updates on the server side, respectively. The global model is periodically evaluated on a held-out central test set. Client training is parallelized using Python `multiprocessing`, where each worker process loads the shared FEMNIST training dataset once and trains a single client model, enabling efficient simulation of a large number of federated clients.

IV. Implemented Algorithms

- **FedAvg (Baseline)**:
  FedAvg serves as the baseline federated learning algorithm in our experiments. Each selected client receives the current global model from the server and performs local training using Stochastic Gradient Descent (SGD) on its private data for a fixed number of local epochs. After local training, each client sends its updated model parameters to the server.
  On the server side, the global model is updated by performing a weighted average of the client models, where the weight of each client is proportional to the size of its local dataset. Let w_t denote the global model at round t, and let $w_t^{(k)}$ be the model of client k after local training. If n_k is the number of samples on client k and $n = \sum_k n_k$ is the total number of samples across participating clients, the server update rule is:

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_t^{(k)}.$$

  In our implementation, the `FedAvgClient` class performs local SGD updates with a cross-entropy loss, while the `FedAvgServer` class aggregates the client state dictionaries using this weighted averaging rule and periodically evaluates the global model on a central test set.

- **FedAdam**$_{v2}$:
  This variant represents a server-optimizer-based extension of FedAvg that applies the Adam optimizer at the server side, using a modified formulation (referred to here as Method 2). As in FedAvg, each client performs standard SGD-based local updates and sends the resulting model parameters back to the server. No adaptive optimization takes place at the client.
  Upon receiving client updates, the server computes the aggregated update and applies the Adam optimizer directly to the global model parameters. Adam maintains first- and second-order moment estimates of past gradients, providing an adaptive per-parameter learning rate that helps stabilize training, especially in heterogeneous or noisy federated environments.
  The Adam update rules applied at the server are:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$w_{t+1} = w_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \tau}$$

  where:
  - g_t is the aggregated gradient (or update) from the clients
  - m_t and v_t are exponential moving averages of first and second moments
  - β_1, β_2 are decay factors for momentum and variance
  - η is the server learning rate
  - τ is a small smoothing term to prevent division by zero

  Compared to standard FedAdam, this version may differ in how gradients or model deltas are preprocessed before Adam is applied, but retains the same adaptive and momentum-based update logic.

- **FedRMSProp**:
  FedRMSProp is a server-optimizer-based federated variant that applies RMSProp during the global model update stage. The clients behave exactly as in FedAvg — performing standard SGD-based local training and sending their updated parameters back to the server. The distinction lies entirely in how the server aggregates and updates the model.
  Instead of direct weighted averaging, the server treats the aggregated model update as a gradient and applies the RMSProp update rule, which maintains an exponential moving average of squared gradients. This makes it particularly useful for handling noisy or unstable updates, which are common in heterogeneous federated settings.
  The server update can be written as:

$$v_t = \alpha v_{t-1} + (1 - \alpha) g_t^2$$

$$w_{t+1} = w_t - \eta \cdot \frac{g_t}{\sqrt{v_t + \tau}}$$

where:

- g_t is the aggregated update from clients at round t
- v_t is the exponentially smoothed squared gradient
- α is the smoothing constant
- η is the server learning rate
- τ is a small numerical stability term

Like FedAdagrad, this method adapts the update magnitude per parameter, but does so using an exponential moving average rather than a cumulative sum. This often leads to better performance in scenarios where client updates are sparse or highly variable.

- **FedAdagrad**:
  FedAdagrad extends the FedAvg framework by incorporating the Adagrad optimizer at the server side. Clients locally train their models using standard SGD, as in FedAvg, and send their parameters to the server. Instead of applying a single weighted average of client updates, the server maintains a per-parameter accumulator of squared gradients and adapts the effective learning rate accordingly.
  This allows parameters that have been updated frequently to receive smaller updates over time, while parameters that are rarely updated receive relatively larger updates. As a result, FedAdagrad can be particularly beneficial in federated settings with sparse or imbalanced client updates, helping stabilize convergence under non-IID data distributions.
  Mathematically, the server update rule is:

$$G_t = G_{t-1} + g_t^2$$

$$w_{t+1} = w_t - \eta \cdot \frac{g_t}{\sqrt{G_t + \tau}}$$

where:

- g_t is the aggregated gradient from clients at round t
- G_t is the accumulated squared gradient
- η is the server learning rate
- τ is a small constant to prevent division by zero

Compared to FedAvg, this adaptive scaling can lead to improved training stability and better performance in highly heterogeneous federated environments.

- **FedAdamW**:
  FedAdamW is a server-optimizer-based variant of FedAvg that applies the AdamW optimizer at the server side. Clients are identical to FedAvg clients: each selected client receives the current global model, performs local training with SGD on its private FEMNIST partition, and returns the updated parameters and local sample count to the server.
  On the server, the aggregated client update is treated as a gradient and AdamW is used to update the global model. AdamW decouples weight decay from the gradient-based update, combining adaptive moment estimation with explicit ℓ_2 regularization on the parameters. Let g_t denote the aggregated gradient at round t, and m_t, v_t be the first and second moment estimates. The AdamW-style update applied by the server can be written as:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\tilde{w}_t = w_t - \eta \frac{m_t}{\sqrt{v_t} + \tau}, \qquad w_{t+1} = \tilde{w}_t - \eta \, \lambda \, w_t,$$

where η is the server learning rate, τ is a small numerical stability term, and λ is the weight decay coefficient. In our experiments, FedAdamW tends to be more sensitive to hyperparameters and non-IID data than FedAvg and FedAdam_V2, often requiring careful tuning of η and weight decay to avoid degraded performance.

- **FedAMSGrad**:
  FedAMSGrad is a server-side adaptive optimization variant of FedAvg derived from the AMSGrad extension of Adam. As in FedAvg, each participating client performs local SGD training and transmits its updated model parameters to the server. No adaptive optimization occurs on the client side.
  At the server, the aggregated client update is treated as a gradient and AMSGrad is applied to update the global model. AMSGrad modifies Adam by maintaining a non-decreasing second-moment estimate, which guarantees convergence in certain non-convex settings where Adam may fail. This is achieved by tracking the element-wise maximum of all historical second-moment values.
  The update rule is given by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t), w_{t+1} = w_t - \eta \cdot \frac{m_t}{\sqrt{\hat{v}_t} + \tau},$$

where η is the server learning rate, β_1 and β_2 are momentum coefficients, and τ is a small numerical stabilizer.
In practice, we find that FedAMSGrad is more stable than FedAdamW under certain settings, but it does not consistently outperform FedAdam_V2 or FedAvg. This suggests that the theoretical stability advantage of AMSGrad does not always translate into improved performance in federated, non-IID environments.

V. Results and Analysis

In this section we evaluate six federated optimization algorithms: *FedAvg* (baseline), *FedAdagrad*, *FedAdam (V2)*, *FedAdamW*, *FedAMSGrad* and *FedRMSProp*. All methods are trained on the FEMNIST dataset with the CNN model described in Section IV, using non-IID client partitions generated via a Dirichlet distribution with concentration parameter $\alpha \in \{0.5, 1.0\}$. Each experiment runs for 30 communication rounds; at every round we record test accuracy and test loss on a held-out server-side test set.

A. Best Configuration for Each Algorithm

Table 1 summarizes, for each algorithm, the best configuration discovered in our hyperparameter search in terms of final test accuracy. We vary the server learning rate, local epochs, batch size and α, while keeping the client optimizer as SGD.

Best Configuration for Each Algorithm

Algorithm	Best Accuracy (%)	Final Loss	Learning Rate	Local Epochs	Batch Size	Alpha
FEDADAGRAD	85.03	0.4567	0.01	3	32	1.0
FEDADAM_V2	86.54	0.3919	0.01	3	32	1.0
FEDADAMW	76.91	1.1773	0.01	3	32	0.5
FEDAMSGRAD	76.69	1.3585	0.01	3	32	1.0
FEDAVG	87.15	0.3646	0.01	3	32	1.0
FEDRMSPROP	86.79	0.3797	0.01	3	32	1.0

Fig. 1. Summary of best configurations for each optimizer.

Overall, FedAvg remains a very strong baseline, achieving the highest final accuracy in our search. However, several adaptive methods, in particular FedAdam_V2 and FedRMSProp, come very close and often converge substantially faster in the early rounds.

B. High-Level Observations

Across all plots (training loss and test accuracy vs. communication rounds) we observe the following consistent patterns:

1) **Two-phase behaviour.** Most methods exhibit a fast initial improvement in the first 5–10 rounds, followed by a slower, more incremental phase. The first phase is dominated by rapid reduction of large, noisy gradients; the second phase refines the model and mainly affects generalization.
2) **Adaptive methods converge faster early.** FedAdam_V2, FedRMSProp and FedAdagrad reach reasonably high accuracy in fewer rounds, especially under high heterogeneity ($\alpha = 0.5$) and when each client performs only a small amount of local work (e.g. one local epoch).
3) **FedAvg often achieves the best final accuracy.** While FedAvg lags adaptive optimizers early on, it typically catches up and surpasses them as the number of rounds increases, particularly for the milder non-IID setting $\alpha = 1.0$ and for reasonably large server learning rates.
4) **AMSGrad and AdamW underperform in this FL setting.** Both FedAMSGrad and FedAdamW show consistently slower convergence and lower final accuracy, unless carefully tuned. AMSGrad tends to be overly conservative, whereas AdamW can be unstable or underfit due to the interaction between decoupled weight decay and non-stationary aggregated updates.
5) **Hyperparameters change the ranking of methods.** The relative performance of the algorithms is highly sensitive to the server learning rate, number of local epochs, batch size and the Dirichlet α. Adaptive optimizers shine when updates are noisy and conflicting, while FedAvg benefits from smoother, more homogeneous updates.

In the remainder of this section we discuss each algorithm in more detail and analyze the effect of the hyperparameters.

C. Per-Algorithm Behaviour

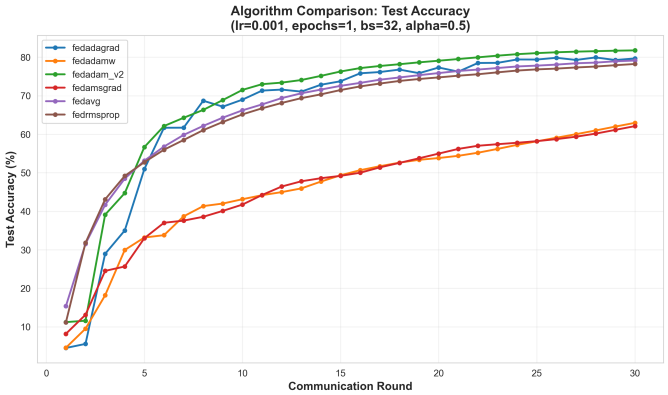

Fig. 2. Test Accuracy vs Rounds

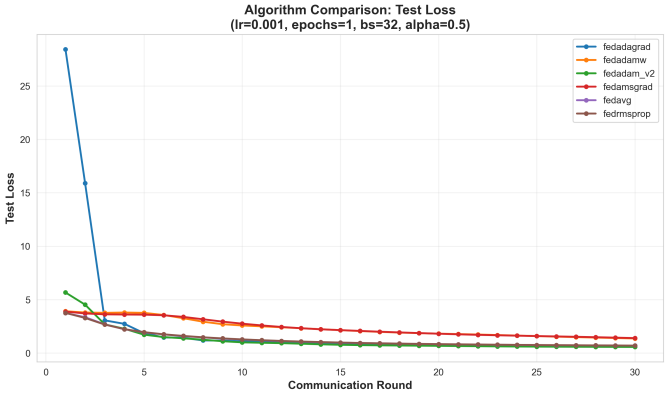

Fig. 3. Test Loss vs Rounds

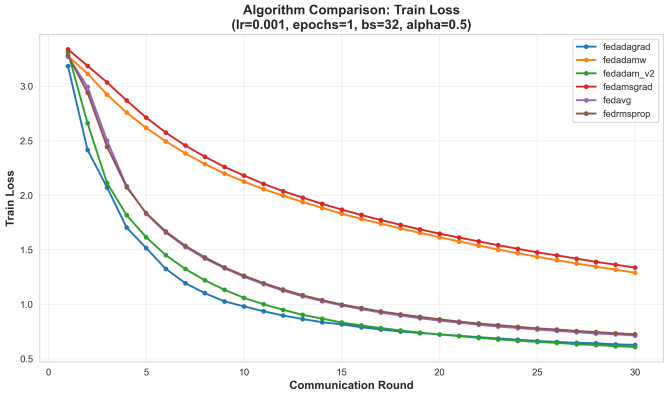

Fig. 4. Train Loss vs Rounds

Fig. 5. Comparison of algorithms under fixed hyperparameters.

1) FedAvg: FedAvg, using pure SGD on clients and weighted averaging at the server, serves as our baseline. In most configurations it exhibits slower early progress than adaptive methods, but often attains the highest final accuracy, as reflected in Table **??**. The loss decreases steadily and the accuracy curve rises more gradually.

This behaviour is consistent with the known properties of SGD: the averaging operation reduces variance across clients, and the inherent stochasticity of SGD encourages exploration of parameter space and tends to favour flatter minima that generalize well. In the early rounds, adaptive methods exploit additional information (such as moment estimates) to move more aggressively, while FedAvg effectively takes smaller, more cautious steps. As the number of rounds grows, the variance of the aggregated updates decreases and FedAvg converges to high-quality solutions. In our experiments, this is particularly visible for $\alpha = 1.0$, where the data is less skewed and the aggregated gradients are already reasonably well aligned.

2) FedAdam_V2: FedAdam_V2 applies the Adam update rule on the server, treating the aggregated client update as a "gradient". This method consistently shows the fastest increase in test accuracy and the steepest early loss reduction, especially in the highly non-IID setting ($\alpha = 0.5$). The exponential moving average of past updates (first moment) acts as a form of server-side momentum, while the moving average of squared updates (second moment) normalizes the step size per parameter.

In the federated context, client updates often differ significantly in magnitude due to non-uniform label distributions and varying local sample sizes. The adaptive rescaling in Adam mitigates this issue by taking larger steps in directions where gradients are consistently large and smaller steps where gradients are noisy or inconsistent. This explains the rapid early progress of FedAdam_V2. However, the same adaptivity can also lead to oscillations or slight overfitting when the server learning rate is large: moment estimates may lag behind the current gradient distribution, so the optimizer can overshoot as the global model and client data distribution co-evolve.

3) FedRMSProp: FedRMSProp uses exponential averaging of squared gradients, similar to Adam, but without maintaining a first-moment (momentum) estimate. It typically converges slightly slower than FedAdam_V2 but remains more conservative and stable, with very smooth loss curves and accuracy trajectories close to FedAvg and FedAdagrad.

The key effect of RMSProp in this setting is to adjust learning rates per parameter according to recent gradient magnitudes. Parameters that are subject to large, noisy updates receive smaller effective step sizes, which dampens instability due to heterogeneous client updates. Unlike AMSGrad, RMSProp allows the second-moment estimate to decrease again if recent gradients become smaller, so it does not suffer from an overly-conservative asymptotic step size. This explains why FedRMSProp often provides a robust middle ground between the speed of Adam and the stability of FedAvg.

4) FedAdagrad: FedAdagrad accumulates squared gradients over time and scales each parameter's learning rate by the inverse square root of this cumulative sum. In our experiments it performs well in many settings and reaches competitive final accuracies, although it can be somewhat conservative in the later rounds.

The behaviour is intuitive: in federated training some parameters may receive many large updates (for example, those associated with classes that are over-represented on certain clients), while others are rarely updated. Adagrad naturally reduces the step size for frequently updated parameters, preventing them from dominating the optimization process and partially compensating for uneven client contributions. However, because the accumulator in classical Adagrad only grows, the effective learning rate per parameter can shrink too much after many rounds, leading to slower progress in the tail of training.

5) FedAdamW: FedAdamW combines Adam with decoupled weight decay at the server. In our setup it consistently underperforms, showing slower convergence, lower final accuracy, and sometimes mild instability, particularly for $\alpha = 0.5$.

The main reason is the interaction between decoupled weight decay and non-stationary federated updates. The global model is updated by aggregating client updates computed without server-side weight decay; only after aggregation does the server apply an additional shrinkage to the parameters. When the client distributions are heterogeneous, this global shrinkage can partially cancel out meaningful client updates, especially for parameters that are important for certain clients but rarely updated elsewhere. Combined with Adam's adaptive scaling, this can lead to overly small effective weights (underfitting) or to an optimization dynamic that is difficult to tune. Our experiments suggest that AdamW requires careful coordination of weight decay and server learning rate across clients and server to perform well.

6) FedAMSGrad: FedAMSGrad modifies Adam by using the maximum of all past second-moment estimates in the denominator, guaranteeing that the effective learning rate for each parameter is non-increasing. This design provides stronger convergence guarantees in adversarial or ill-conditioned settings, but in our experiments it proves to be overly conservative: FedAMSGrad is consistently slower and reaches lower final accuracies than both FedAvg and FedAdam_V2.

In the federated scenario, gradient variances tend to be very large in early rounds and decrease as the model improves and client models become more aligned. Because AMSGrad locks the denominator to the maximum of these early large variances, the effective learning rates remain small even when updates become more consistent. As a result, FedAMSGrad can take extremely cautious steps in later rounds, which explains the slow loss decay and the relatively low ceiling on test accuracy.

D. Effect of Hyperparameters

1) Learning Rate: We vary the server learning rate between 0.001 and 0.01. A higher learning rate accelerates early training for almost all methods by allowing larger steps in the direction of the aggregated gradients. FedAvg, in particular, benefits substantially from an increased learning rate; with $\text{lr} = 0.01$ it becomes competitive with or superior to most adaptive methods in terms of both convergence speed and final accuracy.

Adaptive optimizers such as FedAdam_V2 and FedRMSProp are more robust to moderately large learning rates due to their adaptive scaling, but they can become unstable when

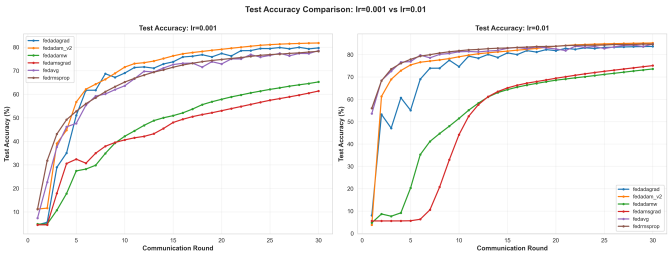

Fig. 6. Test Accuracy vs Rounds

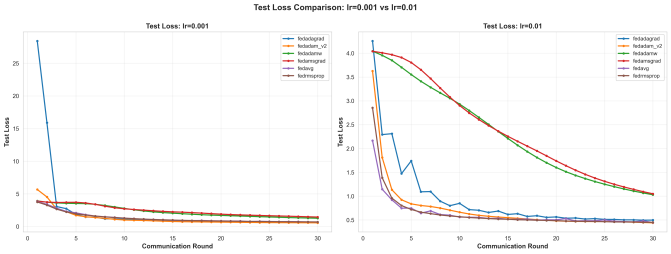

Fig. 7. Test Loss vs Rounds

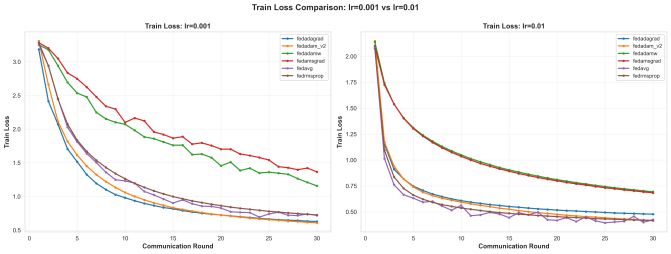

Fig. 8. Train Loss vs Rounds

Fig. 9. Effect of learning rate on each optimizer (0.001 vs 0.01).

the learning rate is too high. In that case the combination of momentum and stale moment estimates can cause the global model to overshoot or oscillate, especially when the aggregated updates are highly non-stationary. Our experiments indicate that a somewhat more conservative server learning rate is desirable for Adam-type methods than for FedAvg.

2) Local Epochs: Increasing the number of local epochs E allows each client to perform more computation per communication round, reducing the overall number of rounds needed for a given level of local progress. However, with non-IID data, larger E also means that client models drift further away from the global model, resulting in more conflicting updates when they are aggregated.

In our experiments, both FedAvg and the adaptive methods generally benefit from increasing E from 1 to 3, in the sense that they reach a given accuracy in fewer rounds. The benefit is particularly clear for FedAvg, which relies on averaging to smooth out local noise; longer local training leads to more informative updates. Adaptive server optimizers such as FedAdam_V2 and FedRMSProp can partially compensate for the increased client drift by downweighting inconsistent directions through their second-moment estimates. Nevertheless, if E is pushed too high, we would expect all methods to eventually suffer due to excessive local overfitting.

3) Batch Size: We consider batch sizes $B = 16$ and $B = 32$. Smaller batches increase the variance of local gradients, making client updates noisier. Adaptive server optimizers

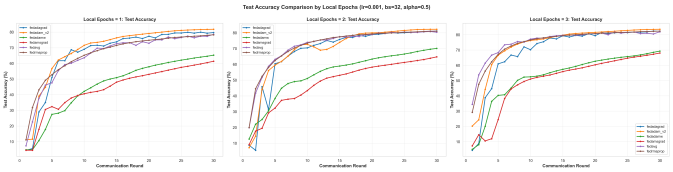

Fig. 10. Test Accuracy vs Rounds

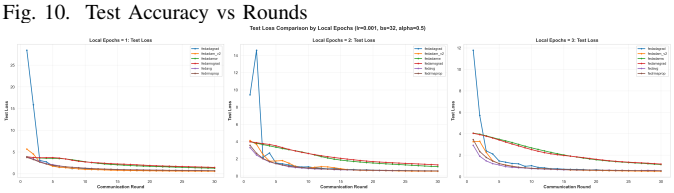

Fig. 11. Test Loss vs Rounds

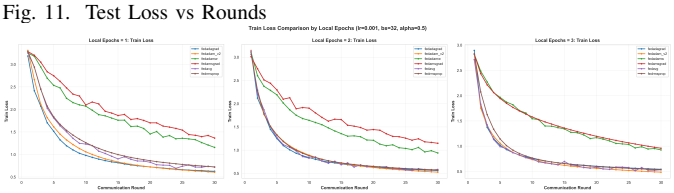

Fig. 12. Train Loss vs Rounds

Fig. 13. Effect of varying local epochs ($E = 1, 2, 3$) on test accuracy and loss.

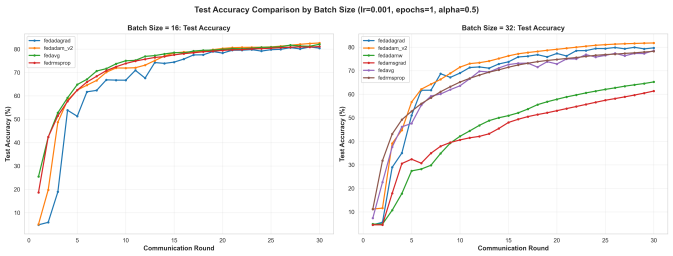

Fig. 14. Test Accuracy vs Rounds

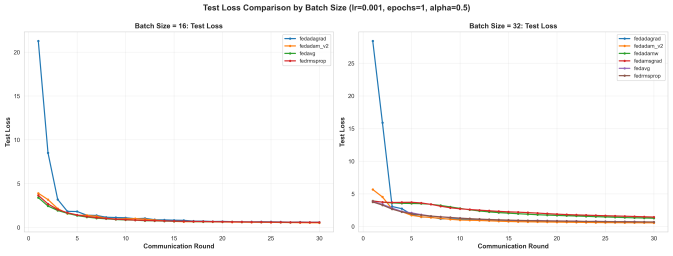

Fig. 15. Test Loss vs Rounds

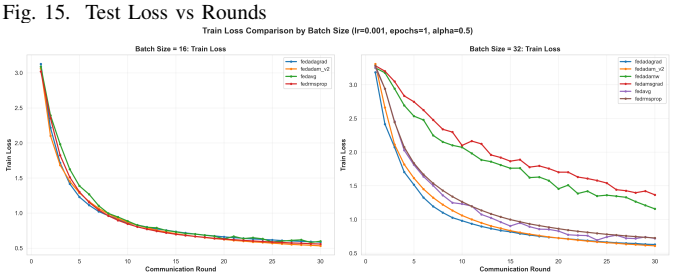

Fig. 16. Train Loss vs Rounds

Fig. 17. Effect of batch size on convergence.

are designed to cope with such variance, and indeed we observe that FedAdam_V2 and FedRMSProp maintain strong performance even for $B = 16$. FedAvg, by contrast, clearly benefits from the larger batch size $B = 32$, which produces smoother local gradients and more stable aggregated updates.

The test loss curves illustrate this behaviour: for $B = 16$ the curves are somewhat noisier, especially in the early rounds, whereas with $B = 32$ the trajectories become smoother and the differences between methods narrow.

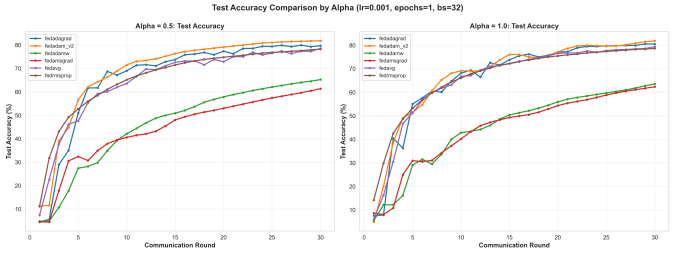

Fig. 18. Test Accuracy vs Rounds

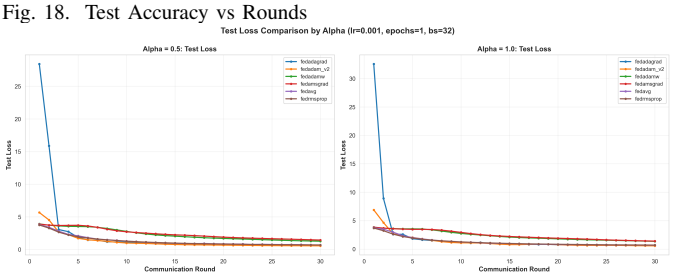

Fig. 19. Test Loss vs Rounds

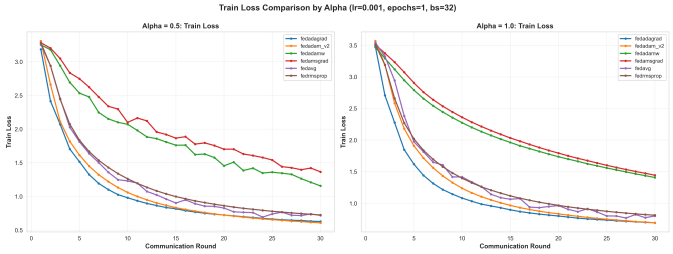

Fig. 20. Train Loss vs Rounds

Fig. 21. Effect of Dirichlet α on optimizer behavior.

4) Data Heterogeneity (Dirichlet α): Finally, we vary the Dirichlet concentration parameter α that controls data heterogeneity across clients. Smaller α values yield more skewed client distributions. For $\alpha = 0.5$, clients often see only a small subset of classes; for $\alpha = 1.0$, the per-client distributions are closer to the global one.

When $\alpha = 0.5$, adaptive methods such as FedAdam_V2 and FedRMSProp provide clear advantages in the early rounds: they converge faster and reach higher accuracy than FedAvg within the same number of rounds. This is because the adaptive rescaling of updates helps to mitigate conflicting gradients arising from non-IID data. However, as training continues, FedAvg gradually catches up as more rounds of averaging reduce variance across clients.

For $\alpha = 1.0$, the difference between methods shrinks. Aggregated gradients are more homogeneous, so the benefits of adaptive scaling are less pronounced. In this regime FedAvg often achieves the best final accuracy, while the adaptive methods remain competitive but no longer clearly superior in terms of convergence speed.

5) Impact of Client Fraction: We next study how the fraction of participating clients in each round affects global convergence. In this experiment, we fix all other hyperparameters to $\text{lr} = 0.001$, epochs $= 1$, batch size $= 32$, and $\alpha = 0.5$, while varying the client fraction in $\{0.5, 0.8, 1.0\}$. A value of 1.0 corresponds to the full-participation setting, where all clients contribute updates in every communication round.

Fig. 22. Test accuracy comparison under different client fractions.

a) Test Accuracy Dynamics.: When only 50% of the clients participate in each round, FedAvg still converges but does so more slowly due to reduced diversity and fewer updates per round. Increasing the client fraction to 0.8 gives substantially faster convergence: adaptive methods benefit more from the increased participation, as they obtain more consistent aggregated signals from the server's optimizer state. At full participation (client fraction = 1.0), all methods improve further. In this setting, FedAdam_V2 reaches the highest accuracy fastest, while FedAvg again catches up and achieves competitive final accuracy.

These trends illustrate that communication efficiency and update diversity interact strongly: more participating clients reduce the variance of global updates, which especially favors adaptive optimizers that depend on stable moment estimation.

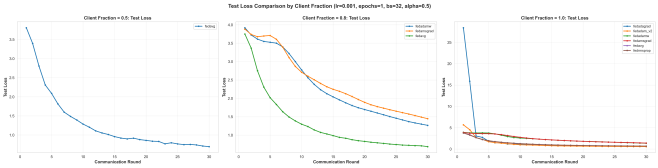

Fig. 23. Test loss comparison under different client fractions.

b) Test Loss Curves.: At low client fraction (0.5), loss decreases smoothly but relatively slowly. With 0.8 participation, we observe accelerated loss decay for FedAvg and FedAdamW, though FedAdamW remains weaker overall. At full participation, test loss curves become smoother and more tightly clustered, indicating better optimization stability. Importantly, FedRMSProp and FedAdam_V2 show rapid early loss reduction when 100% of clients contribute, benefiting from stronger aggregated signal and improved moment estimation.

c) Train Loss Behavior.: Train loss curves follow a similar pattern: increasing the client fraction leads to faster and more consistent loss reduction. Interestingly, when the fraction is low (0.5), FedAvg is still able to optimize steadily, which reflects the robustness of averaging-based updates even

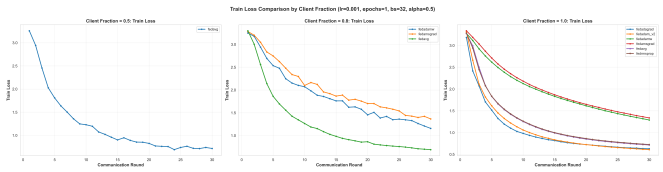

Fig. 24. Train loss comparison under different client fractions.

under missing information. However, adaptive methods such as FedAdam_V2 and FedRMSProp clearly benefit more from the higher client participation rates, as they rely on consistent aggregated updates for stable moment tracking.

d) Interpretation.: These results highlight a key aspect of federated optimization:

- **Lower client fractions reduce the richness of update information, slowing convergence.**
- **Adaptive methods rely more heavily on consistent, stable aggregated gradients**, and therefore benefit disproportionately from higher client participation.
- **FedAvg remains relatively stable even under partial participation**, as simple averaging tolerates update sparsity better than moment-based adaptive optimizers.

In summary, increasing the client fraction improves both convergence speed and stability, particularly for adaptive FL algorithms. When communication resources allow, higher participation rates are beneficial for both test accuracy and loss reduction.

E. Discussion and Practical Implications

The experiments highlight a number of practically relevant trends:

- **FedAvg as a strong baseline.** Despite its simplicity, FedAvg remains a very competitive method, especially when tuned with a moderately large learning rate and a few local epochs. It often achieves the best final test accuracy, which is consistent with the favourable generalization properties of SGD.
- **Adaptive optimizers for fast early progress.** FedAdam_V2 and FedRMSProp are attractive when the goal is to obtain good performance in a limited number of communication rounds or when data heterogeneity is high. Their moment estimates and adaptive learning rates allow them to exploit information from noisy, conflicting client updates more effectively.
- **Caution with AdamW and AMSGrad.** FedAdamW and FedAMSGrad are more difficult to tune in the federated setting and underperform in our experiments. AdamW's decoupled weight decay can counteract useful client updates, while AMSGrad's non-increasing step sizes make it excessively conservative once the variance of gradients decreases.
- **Hyperparameter sensitivity.** The relative ranking of algorithms is not fixed: it depends strongly on the learning rate, local epochs, batch size and α. In practice, a small hyperparameter sweep around these values is essential before drawing conclusions about the superiority of a given optimizer.

Overall, our results suggest the following practical guideline: if communication is cheap and the objective is to maximize final accuracy, FedAvg with a carefully chosen learning rate and a few local epochs is a robust default choice. If communication is expensive or rapid progress is required under strong heterogeneity, adaptive server optimizers such as FedAdam_V2 or FedRMSProp are preferable, provided that their hyperparameters are tuned conservatively to avoid instability.

APPENDIX

To illustrate the effect of non-IID partitioning, Figure 25 shows the class-wise data distribution for 10 randomly sampled clients, generated using a Dirichlet distribution. Each subplot corresponds to one client and displays the number of samples for each of the 62 FEMNIST classes.

As seen in the plots, certain clients possess data heavily skewed toward a small subset of classes, reflecting realistic statistical heterogeneity. This setup is crucial for evaluating how different federated optimization algorithms behave under non-IID settings.

Fig. 25. Data distribution across clients under Dirichlet-based non-IID split. Each subplot shows label frequencies for one client.