EE5102/CS6302 - Advanced Topics in Machine Learning – Fall 2025

Assignment 4 — *Release Date: 2 November 2025*

Instructions: Please read the following instructions carefully and abide by while preparing your submissions:

- This is the fourth graded assignment of the course which counts towards 7% of your final assignment aggregate.
- We need only one submission per group. If you submit the report for your only task and/or not in the required format, we will assign you zero marks. **Note: Task 4 is heavy.** .
- Each member is recommended to do one task from Task 1, 2 and 3, and atleast one sub-part of Task 4.
- This assignment is due by **Friday, 14 November 2025 on LMS**.
- As a submission, you need to prepare and submit you deliverables according to the instructions in **Task 4 of this assignment**.
- **AI Usage Policy:** You are not allowed to use any generative AI model—including LLMs—to write any part of your PDF report. The language/text and analysis must be entirely your own, in the report.

  For the coding part, you ***may*** use public libraries, built-in functions, or even get help from an LLM if needed. That's acceptable.

  However, once you submit your assignment, you are fully responsible for everything in it—text and code both. If your submission overlaps significantly with another group's work, you cannot later claim that some part was LLM-generated. That will not be accepted as an excuse.

Overview & Motivation: Federated Learning (FL) is a decentralized machine learning paradigm where multiple clients collaboratively train a shared global model without exchanging their private data. Instead of centralizing datasets, each client C_i keeps its local dataset D_i and periodically exchanges model parameters or updates with a central server. The goal is to minimize the global loss:

$$L(\theta) = \sum_{i=1}^{M} \frac{N_i}{N} L_i(\theta),$$

where $L_i(\theta)$ is the local loss on client i, $N_i = |D_i|$ is the size of client i's data, and $N = \sum_{i=1}^{M} N_i$ is the total data size across all M clients. The term N_i/N acts as a weight so that each client's contribution is proportional to its data fraction in the overall network.

Federated optimization process: Training proceeds over multiple communication rounds $t = 1, 2, \ldots, T$. In each round, the server holds the current global model θ_g^t. The typical sequence is:

- **Broadcast:** The server sends the current global model θ_g^t to some or all clients.
- **Local Training:** Each selected client C_i initializes its local model $\theta_i^{(0)} = \theta_g^t$ and performs K steps of local optimization (e.g., Stochastic Gradient Descent) on its own data D_i. After K local epochs (or mini-batch iterations), the client obtains an updated model $\theta_i^{(K)}$.
- **Upload:** Each client sends its update (either the new weights $\theta_i^{(K)}$ or the difference from the initial model) back to the server.
- **Aggregation:** The server computes a weighted average of the received updates to form the next global model θ_g^{t+1}. Commonly, the server uses weight N_i/N for client i so that

  $$\theta_g^{t+1} = \sum_{i=1}^{M} \frac{N_i}{N} \theta_i^{(K)}$$

  (assuming all selected clients responded).

Two important special cases of this procedure are FedSGD and FedAvg:

Federated SGD (FedSGD): Each client performs $K = 1$ local step (typically one mini-batch SGD step) per round. If all clients participate in every round and use the same learning rate as a centralized scheme, FedSGD's update is equivalent to a single global SGD step on the combined data. In other words, averaging one-step gradients from each client yields the same update as computing the gradient on all data at once. FedSGD is theoretically equivalent to centralized mini-batch SGD in this ideal setting (same initial model, IID data, full participation), but it is communication-inefficient because it requires a communication round for every single SGD step.

Federated Averaging (FedAvg): Proposed by McMahan et al., FedAvg improves efficiency by letting each client do $K > 1$ local SGD steps (or epochs) before syncing. This drastically reduces the number of communication rounds needed for a given amount of local computation. However, as K increases (especially under non-IID data), clients may drift towards different local optima, causing the local models θ_i^* to diverge from the true global optimum θ_g^*. This phenomenon is known as *client drift*, and it can slow down or destabilize convergence in heterogeneous settings. FedAvg is the de-facto baseline for FL, but its performance degrades when client data distributions are highly divergent.

Key challenges in FL:

- **Privacy:** The system must ensure that sharing model parameters or gradients does not leak private information about any client's data. (This is a motivation for FL itself, but beyond the scope of this assignment).
- **Communication Cost:** Sending large models repeatedly can be expensive (latency, bandwidth). FL algorithms strive to minimize rounds of communication or the size of transmitted updates.
- **Heterogeneity:** Perhaps the most crucial challenge — clients may have heterogeneous data distributions (statistical heterogeneity) and varying system capabilities (device heterogeneity). Statistical heterogeneity means $P_i(x, y) \neq P_j(x, y)$ for data on clients $i \neq j$ (non-IID data), for example one client's data might be mostly a certain class (label skew) or drawn from a different input distribution (feature skew). Device heterogeneity refers to differences in hardware (CPU/GPU, memory), network quality, battery, etc., which can cause some clients to train faster or slower, or drop out.

This assignment will focus on statistical heterogeneity (non-IID data) and methods to mitigate its impact on FL training. Non-IID data leads to inconsistent local optima across clients: each client's local objective $L_i(\theta)$ is minimized at some θ_i^*, but those optima differ, so the aggregated model may not be optimal for any client. One way to quantify heterogeneity is via the bounded client dissimilarity assumption, which bounds how far individual gradients can stray from the global gradient:

$$\sum_{i=1}^{M} \frac{N_i}{N} \|\nabla L_i(\theta)\|^2 \leq G^2 + B^2 \|\nabla L(\theta)\|^2.$$

FedProx (Federated Proximal): FedProx adds a proximal term to the client objective to keep local models closer to the global model. This helps prevent local models from drifting too far. FedProx is a generalization of FedAvg with a tunable parameter $\mu \geq 0$ controlling the strength of this regularization. FedProx improves convergence stability under heterogeneity, but if μ is too large it can bias the solution away from the true global optimum (since clients are overly constrained).

FedDyn (Federated Dynamic Regularization): FedDyn introduces an adaptive regularizer at each client to exactly counteract the difference between local and global objectives. The server maintains an auxiliary variable (essentially tracking the global gradient history) that is used to modify each client's loss function such that, in the limit, local minima align with the global minimum. FedDyn provably eliminates stationary bias (no drift in the limit) and was shown to converge faster than FedAvg in heterogeneous settings (ICLR 2021).

SCAFFOLD (Stochastic Controlled Averaging): SCAFFOLD uses control variates (variance reduction techniques) to correct client drift. The server and clients maintain additional variables c (global control) and c_i (client-specific control) that estimate the global gradient. During local training, clients adjust their gradients using these control variates to stay closer to the global optimization direction. SCAFFOLD theoretically removes the effect of data heterogeneity (convergence is unaffected by B in the above equation) and often requires significantly fewer rounds than FedAvg. The trade-off is that the server must send and receive the extra c vectors in each round, roughly doubling communication per client.

FedGH (Gradient Harmonization): FedGH (Gradient Harmonization) addresses drift at the server aggregation step by detecting and reducing gradient conflicts. A gradient conflict means that two clients' model updates (gradients) are pointing in conflicting directions (their dot product is negative). FedGH (Federated Gradient Harmonization) computes the cosine similarity between every pair of client gradients; if it finds a pair with cosine

< 0, it projects each gradient onto the orthogonal complement of the other to remove the conflicting component. By harmonizing updates this way before averaging, the global update is not pulled in opposing directions, mitigating the effect of divergent local objectives. FedGH has been shown to significantly improve accuracy in highly non-IID scenarios, as it consistently boosts FedAvg and other methods by 1.5–5% on image classification benchmarks, with larger gains when heterogeneity is more severe. The downside is increased computation on the server (pairwise projections) and the requirement that clients transmit their gradient or model update (which is usually the case in FL).

FedSAM (Sharpness-Aware Minimization in FL): FedSAM integrates the idea of Sharpness-Aware Minimization (SAM) into local training. Standard training (Empirical Risk Minimization) can lead to models that converge to sharp minima, especially when each client overfits its local data. SAM (foremost introduced by Foret et al., 2021) seeks flat minima by modifying the update: for each gradient step, it first finds a perturbation in model weights that maximally increases the loss, then it updates the model to minimize the loss at that perturbed point. In an FL context, FedSAM means each client, instead of ordinary SGD, uses SAM for its local updates. This tends to improve the generalization of the global model and reduce client drift, as clients don't over-optimize sharply on their own data. Variants like FedGMT, FedLESAM, FedGAMMA further refine this by reducing SAM's overhead or aligning perturbations to the global landscape (e.g. FedLESAM finds an approximate global perturbation direction to use on each client, requiring only one backward pass per update). In general, FedSAM and its variants significantly boost test accuracy in non-IID settings, though at the cost of more local computation per round.

In this programming assignment, you will implement and experiment with these federated learning algorithms using PyTorch. The tasks will guide you through:

- Verifying the equivalence of FedSGD to centralized training in a simple scenario.
- Implementing FedAvg and quantifying the impact of label distribution skew (a type of non-IID data) on model performance.
- Extending your FedAvg implementation with four advanced techniques (FedProx, SCAFFOLD, Gradient Harmonization, and FedSAM) to evaluate how each addresses the heterogeneity problem.

Note: standard FL benchmarks in literature include CIFAR-10, CIFAR-100, EMNIST/FEMNIST, Shakespeare (text), and synthetic non-IID data. For our purposes, CIFAR-10 is a good choice; it's widely used in many FL papers, e.g., FedAvg and FedProx. MNIST (handwritten digits) or FMNIST (fashion MNIST) are simpler alternatives if you have very limited compute, though these are somewhat easier tasks. Use 5–10 clients for most experiments (unless otherwise specified) – this is enough to observe heterogeneity effects without being too slow, and aligns with a cross-silo FL scenario where we have only a handful of clients. All clients will share the same model architecture e.g. a small convolutional network with a couple of conv layers and a small FC layer, as used in many papers. Clearly mention the architecture of your model first and then start the Tasks.

Finally, ensure you implement with clear structure and documented code. You will likely need to refactor your code to avoid duplication since many algorithms share components. For example, you might write a generic federated training loop and pass in a strategy-specific hook or parameter (like a function for local training or an aggregation modifier).

Now, let's proceed with the tasks.

Task 1: FedSGD vs Centralized SGD (Theoretical Equivalence) The first task is to demonstrate the theoretical equivalence between FedSGD and centralized training under a controlled scenario. This will help you build intuition and verify your understanding of federated optimization.

In theory, if (a) all clients participate in every round, (b) each client computes the gradient of the loss on its entire local dataset (i.e. one full pass, or at least a gradient on all local data) and applies one step of SGD, and (c) we use the same global learning rate as an equivalent centralized SGD, then performing one communication round of FedSGD is mathematically the same as one step of mini-batch SGD on the union of all data. Essentially, FedSGD with full participation and one local step per round averages the individual gradients: $\theta_g^{t+1} = \theta_g^t - \eta \sum_{i=1}^{M} \frac{N_i}{N} \nabla L_i(\theta_g^t)$, which equals $\theta_g^t - \eta \nabla L(\theta_g^t)$ because $L(\theta) = \sum_i \frac{N_i}{N} L_i(\theta)$. This is exactly the update rule of centralized gradient descent on $L(\theta)$.

Now, implement a simple simulation of FedSGD and compare it to centralized SGD:

- Choose a toy dataset for which you can easily control the distribution across clients. For example, take a subset of MNIST or CIFAR-10 and split it evenly into $M = 3$ or 6 clients with IID data (each client gets a random sample of the data). Ensure each client's data size is the same for simplicity.
- Use a small model (e.g. small 2 or 3-layer neural network) so that you can precisely track weight updates.
- Configure the training such that each FedSGD round = one SGD step per client on their local data:
  - Each client computes the gradient on its whole local set (or you can simulate this by using a single batch that is the entire local dataset).
  - The server aggregates the gradients (or weight deltas) and updates the global model.
- In parallel, set up an equivalent centralized training run:
  - Combine all client data into one dataset.
  - Use the same initial model and same learning rate η.
  - In each step, compute the gradient on the entire batch of combined data and update the model.
- Run both for a few steps (say 10 to 20 iterations), ensuring that in each iteration the total amount of data used is the same (FedSGD uses all data split on clients, central uses all data at once).
- Verify the parameter updates: After each iteration, compare the global model weights from FedSGD vs centralized. They should be nearly identical (up to minor differences due to floating point or ordering) if conditions were met. You can print out the norm of the difference between the two models after each round to quantify any discrepancy.
- Compare training progress: Track a simple metric like training loss or accuracy on a validation set for both approaches. They should progress in sync. For example, if you train a softmax classifier on MNIST for 5 FedSGD rounds (with all clients each doing one full-batch update) versus 5 centralized SGD steps (with the full data), the loss and accuracy trajectories should match.

Controlled scenario: To truly satisfy the theoretical assumptions, you might use a tiny learning rate and convex loss to avoid any randomness. However, even with typical settings for a neural network, FedSGD and centralized should behave equivalently on IID partitions. If you notice a difference, double-check that all clients were indeed included each round and that no extra local steps crept in.

Expected outcome: You should observe that FedSGD exactly mirrors centralized SGD in this ideal case. This demonstrates that any differences in FedAvg's behavior (Task 2 onwards) are due to more than just the distributed nature – they arise from additional local steps (when $K > 1$) and from data heterogeneity.

The code implementation for the FedSGD process (you can integrate this with your later FedAvg code, since FedSGD is a special case of FedAvg with 1 local epoch). A short explanation showing that the model parameters or metrics for FedSGD and centralized training remain very close or identical through the iterations. Also, briefly explain why they match, confirming your understanding of the gradient equivalence.

Task 2: Implementing FedAvg: In this task you will implement the **FedAvg** algorithm (with $K > 1$ local epochs per round) and use it to study how we can reduce communication overhead in federated learning. The first strategy is increasing local work per round by raising the number of local epochs/updates K between synchronizations. The second is client sampling or partial participation by aggregating updates from a random n out of N clients each round (vary n depending on available bandwidth). The goal is to measure performance (global test accuracy over communication rounds) and stability (convergence behaviour / oscillations) for both strategies and present **two primary plots**: one comparing different K values, and one comparing different client sampling fractions n/N.

- **Implement FedAvg:** Build upon your FedSGD code. You should repeat the same toy datasets (e.g CIFAR-10) and toy models (e.g 2-3 Layer CNNs) from the previous task. Key implementation points for FedAvg include:
  - Each client will run $K > 1$ local epochs (or a certain number of mini-batch updates) on its own data per round. In practice, choose a small K (like 5) to start, to avoid too aggressive divergence. Later, you can experiment with different K.
  - After local training, each client sends either the new model parameters $\theta_i^{(K)}$ or the weight delta ($\Delta_i = \theta_i^{(K)} - \theta_g^t$). The server averages them: $\theta_g^{t+1} = \sum_i \frac{N_i}{N} \theta_i^{(K)}$ (if using parameters).
  - Ensure your implementation accounts for the weighting by data size. If all clients have equal data, simple average is fine; otherwise do a weighted average.
  - Use PyTorch to handle model parameters and optimization. A convenient approach is to perform local training using standard PyTorch optimizers on each client model, then manually average the model weights (tensors) for aggregation.
- **Experimental Setup:**
  - For the first scheme, start by choosing different values of K, e.g $K \in \{1, 5, 10, 20\}$, and set $S_t = \{1, \ldots, N\}$ (all clients in each communication round).
  - For the second scheme, vary the sampling fraction while keeping K small: choose sampling fractions $f \in \{1.0, 0.5, 0.2\}$ (i.e. $n = fN$ clients sampled uniformly without replacement each round) and keep K fixed (e.g. $K = 1$ or $K = 5$). This isolates the effect of partial participation.
  - Evaluate both schemes for performance and stability.
  - **Evaluate global model's performance:** After each communication round t (post aggregation), evaluate the *global* model θ_g^t's accuracy on a held-out test set.
  - **Measure client drift or divergence:** In addition to accuracy, you can quantitatively measure the divergence of client models from the global model during training by calculating weight divergence:

$$d_\theta^t = \frac{1}{M} \sum_i \|\theta_i(t, K) - \theta_g^t\|,$$

    the average distance between client i's model (after local training in round t) and the global model before aggregation. Larger values indicate more drift in that round.

    Another metric is regret or how much local objectives are minimized relative to the global: e.g.

$$R_g^t = L(\theta_g^t) - \min_\theta L(\theta)$$

    (if $\min L(\theta)$ is known or approximated).

You will observe and report how K and f affect accuracy, convergence, and model drift. Higher K lowers synchronization frequency but can cause instability, while smaller f reduces communication per round but may slow or destabilize training (i.e increasing client drift). You'll plot accuracy vs. rounds for both variations. Also analyze the final accuracy, drift, and communication cost to clearly highlight the trade-offs between efficiency and performance.

Task 3: Exploring Data Heterogeneity Impact: In this task, you will reuse your implementation of FedAvg from Task 2 with a fixed K, and use it to study how label heterogeneity affects model performance. Label heterogeneity (or label skew) means different clients see different distributions of labels (classes). For instance, one client might have mostly class A and B, while another has mostly class C, etc. This is a common type of non-IID data that greatly challenges federated training.

- **Simulate heterogeneous data with Dirichlet distribution:** To systematically vary the degree of non-IIDness, use a Dirichlet distribution to partition the dataset by class. The Dirichlet partitioning method is widely used in FL research to create controllable label skew.
  - For a given concentration parameter $\alpha > 0$, drawing a sample from a Dirichlet $\text{Dir}(\alpha)$ across M clients will produce a probability vector for each class, determining how that class's samples are divided among the M clients.
  - Small α (e.g. 0.1) yields highly skewed distributions (each client gets mostly a few classes), whereas large α (e.g. 100, or $\to \infty$) yields almost uniform distribution (IID case).
  - Use a library or write your own function to sample class proportions for each client using $\text{Dir}_M(\alpha)$. Then assign images to clients accordingly. (Make sure each client gets a sufficient number of samples of each class it is assigned; you might need to sample in a round-robin or proportional way to match the exact counts.)
- For example, you can try $\alpha \in \{0.05, 0.2, 1, 100\}$:
  - $\alpha = 100$ approximates IID (each client $\sim$10% of each class for 10 classes).
  - $\alpha = 1$ is moderately heterogeneous.
  - $\alpha = 0.2$ or 0.05 are highly skewed (clients mostly see distinct classes).
- **Experiment settings:** Use a common model across clients (e.g. a simple CNN suitable for CIFAR-10: two convolutional layers + pooling + a couple of dense layers). Ensure the model is small enough to train reasonably on a single GPU/CPU with 5–10 clients. Set a fixed number of total communication rounds (for instance, 50 rounds) and a fixed local epoch count K (like 5 local epochs per round) for all runs. Also decide on a learning rate (and possibly use a scheduler or decay if needed). A typical setting might be: 5 clients, each with 10 local epochs, 50 rounds, learning rate 0.01 SGD. Make sure to hold all hyperparameters constant except the data distribution when comparing runs.
- **Evaluate global model performance:** At the end of each round (after aggregation), compute the global model's accuracy on a held-out test set (e.g. the standard CIFAR-10 test set). This gives the global model accuracy over time.
- **Vary the heterogeneity (Dirichlet α):** Conduct multiple runs of training with different α values to produce different levels of label skew. For fairness, use the same total number of training rounds and same initialization for each run if possible (to isolate the effect of data distribution).
- **Plot and observe the impact:** Plot the test accuracy versus communication rounds for each scenario (you can plot accuracy on the y-axis and round on the x-axis, with separate curves for each α setting). You should observe a clear trend: as the data distribution becomes more skewed (smaller α), the final accuracy drops and convergence may slow or become unstable. For example, you might see that with IID data ($\alpha = 100$ or very large) the model reaches say $\sim$80% accuracy, but with $\alpha = 0.1$ it only reaches $\sim$50–60%. Also, the non-IID runs might have more oscillation in the accuracy curve due to clients alternating between disparate local optima.
- **Measure client drift or divergence:** Just like Task 2, You can quantitatively measure the divergence of client models using weight divergence or regret.

Expected observations:

- As label heterogeneity increases (lower α), FedAvg performs worse. This manifests as lower final accuracy and possibly slower convergence. This is a known behavior: FedAvg can even fail to converge in extreme non-IID cases because local updates move in very different directions.
- For moderately large α (nearly IID), FedAvg should perform close to centralized training (which is the upper bound).
- You might notice that the more skewed runs have not only lower accuracy but also the gap between training accuracy (which you can measure per client) and global test accuracy grows – indicating the model fits some clients well but doesn't generalize globally.

Make sure to plot the results clearly (with legends indicating the data distribution of each curve). Also record the final accuracy after a fixed number of rounds for each scenario, as a summary. You can plot global test accuracy vs rounds for different non-IID settings. (Ensure axes are labeled and units clear). A short description of results: how does increasing non-IIDness affect performance? Does it match your expectation that "as label skew increases, performance typically decreases"? Provide concrete numbers from your experiments to back this up. If you computed any drift metrics or other observations (like perhaps one client consistently lags or its data dominates), note that as well.

Keep the compute constraints in mind: using too many rounds or too complex a model on CIFAR-10 can be slow. You can start with fewer rounds (e.g. 20) to see the trend, and later extend if time allows. Also consider using partial client participation if needed – e.g. sample 5 out of 10 clients each round – but for this analysis, it's clearer if all clients participate every round.

Task 4: Mitigating Heterogeneity – FedAvg Extensions: Having seen the detrimental impact of non-IID data on FedAvg, the next step is to implement and experiment with several improved federated optimization techniques. We will focus on four categories of approaches discussed in the background: FedProx (regularization), SCAFFOLD (control variates), Gradient Harmonization (FedGH), and Sharpness-Aware Minimization (FedSAM). Each method tries to address the heterogeneity challenge in a different way. You will implement each and evaluate its performance under varying data heterogeneity, comparing against the FedAvg baseline from Task 2.

It's recommended to implement these as modular extensions of your FedAvg code, rather than writing entirely separate codebases. For example, you might have a flag or parameter that specifies which algorithm to run, and then conditionally execute the extra steps (like adding a regularizer, or adjusting gradients) as needed.

We suggest tackling them one by one:

Task 4.1: FedProx – Local Regularization FedProx (Li et al., MLSys 2020) modifies the FedAvg objective by adding a proximal term $\frac{\mu}{2}\|\theta - \theta_g^t\|^2$ to each client's loss. Here θ_g^t is the global model sent to the client at the start of the round, and θ represents the local model parameters being optimized. Intuitively, this term penalizes the local model for moving too far from the initial global model in that round. When $\mu = 0$, this is just FedAvg; a larger μ forces local updates to stay closer to θ_g^t. This can stabilize training on non-IID data by reducing the variance between local updates.

Implementation:

- In your client update loop, after computing the regular data loss (e.g. cross-entropy on local batches), add the proximal term:

  $$\text{prox\_loss} = \frac{\mu}{2}\|\theta_{\text{local}} - \theta_{\text{global}}\|^2.$$

  In practice, θ_{global} is a fixed tensor (from the round's start) and θ_{local} are the live model parameters. You can compute this efficiently by summing up $\|\theta_i - \theta_{g,i}\|^2$ for all parameters.
- Backpropagate and optimize using this modified loss. Most autodiff frameworks (e.g., PyTorch) will handle the gradient of that L_2 term just fine (it will add $\mu(\theta - \theta_g)$ to each weight's gradient).
- Choose a value for μ. In literature, they often use a small μ like 0.001 or 0.01 for convex problems, sometimes larger (0.1) for deep networks. You might need to tune μ a bit. A too-large μ will make local training ineffective (clients hardly move from the initial model); too small might have no noticeable effect. For initial experiments, try $\mu = 0.01$.
- All other aspects of FedAvg remain the same (you still do K local epochs, etc.). The proximal term only modifies the local update equation.

What to observe: FedProx is expected to improve stability and sometimes the final accuracy in heterogeneous settings compared to FedAvg. However, because it effectively restricts the solution space, it might converge to a slightly worse optimum than FedAvg could if FedAvg were to converge well. In other words, FedProx might trade off a bit of bias for a reduction in variance. In practice, you might see FedProx and FedAvg have similar accuracy on IID or mild non-IID data, but FedProx outperforms FedAvg as heterogeneity increases (FedAvg might fail to converge, whereas FedProx keeps making progress).

Run FedProx on the same experimental setup as Task 2 (use at least one highly skewed scenario, e.g. $\alpha = 0.1$, and one moderate/IID scenario). Compare its test accuracy curve to FedAvg's curve. Record the final accuracy for each. Does FedProx help? How sensitive is it to μ?

You can optionally try a few different μ values to see the effect. Too large μ might underfit, too small is essentially FedAvg. But this parameter tuning is secondary.

Task 4.2: SCAFFOLD – Controlling Client Drift SCAFFOLD (Karimireddy et al., ICML 2020) uses control variates to correct the client drift. The server maintains a global control vector c and each client i maintains a local control vector c_i. These vectors have the same dimension as model parameters (or gradients) and track the difference between the global gradient and the client's local gradient estimations. At a high level, SCAFFOLD tells each client to subtract the drift during local training. If a client's updates tend to go in some direction that is not aligned with the global objective, the control variate steers it back.

Key update rules (high-level):

- Initially, $c = 0$ (global) and $c_i = 0$ for all clients.
- When the server selects clients for a round, it sends them the current global model θ_g^t along with the global control c^t.

- Each client will perform local updates with correction. If w denotes model parameters and $g(w)$ the local gradient on a batch, the update step might look like:

$$w \leftarrow w - \eta\big(g(w) + (c_i^t - c^t)\big),$$

meaning the client adjusts its gradient by adding $(c_i - c)$. (The original paper's Algorithm 1 provides precise detail – essentially, c and c_i act like estimates of the gradient; the difference guides the update).

- After completing K local epochs, the client computes its new control c_i^{t+1}. One way to compute this is:

$$c_i^{t+1} = c^t - \frac{1}{\eta K}(\theta_g^t - \theta_i^{(K)}),$$

i.e., how much did the model change relative to expected if following global grad. (This formula comes from setting the variance of gradient difference to zero; you can derive or find it in the paper.)

- The client sends both its model $\theta_i^{(K)}$ and the updated c_i^{t+1} back to the server.

- The server updates the global model as usual (weighted average of $\theta_i^{(K)}$). It then updates the global control as:

$$c^{t+1} = c^t + \frac{1}{M}\sum_{i \in \text{clients}} (c_i^{t+1} - c^t),$$

which simplifies to

$$c^{t+1} = \frac{1}{M}\sum_i c_i^{t+1}$$

if all clients participated (or a weighted average if not). In effect, the global control becomes the average of all client controls (which makes it an estimate of the true global gradient after convergence).

Implementing SCAFFOLD requires carefully managing these additional vectors. You will likely maintain a `c_global` array of tensors for each model parameter (or one big concatenated vector) and a dictionary of `c_local[i]` for each client.

Implementation tips:

- **Data structure:** `c_global` can be a list of tensors matching your model's `.parameters()`. Same for each `c_local[i]`. Initialize all to zero.

- **Client side:** When a client is about to train in round t, load c_{global}^t and its own $c_{\text{local}}[i]^t$. During each optimizer step on that client, modify the gradient of each weight as:

$$\text{grad} = \text{grad} + (c_{\text{local}}[i] - c_{\text{global}})$$

(for that weight). In PyTorch, you might manually apply this by hooking into the optimizer or by adjusting the `.grad` before `optimizer.step()`.

- **Local update done:** After local training, compute the difference

$$\Delta\theta_i = \theta_g^t - \theta_i^{(K)}.$$

Then update the client's control:

$$c_{\text{local}}[i] = c_{\text{global}} + \frac{1}{K \times \text{lr}}(\Delta\theta),$$

(this is equivalent to the formula above but rearranged).

- **Server side:** After receiving all, update

$$c_{\text{global}} = c_{\text{global}} + \text{average}(c_{\text{local deltas}}).$$

Actually, notice from above: the server update for c_{global} will end up equal to the average of new c_{locals}, which after our update formula should remain the same as old c_{global} if all clients participated and did full local optimization. However, implement it as described for consistency. (If all clients participate, some analyses show c_{global} might remain 0 if started 0 – but with partial participation it evolves.)

- **Re-use as much of FedAvg code as possible:** the main difference is in how gradients are computed and the extra step of sending/receiving c values.

Experiment: Test SCAFFOLD in a challenging non-IID scenario (like $\alpha = 0.1$ or even more skew). Compare its performance to FedAvg and FedProx. SCAFFOLD should do significantly better, often approaching IID performance even with strong heterogeneity. It might converge faster (fewer rounds to reach the same accuracy) and to a higher final accuracy. One thing to check is the communication overhead: you are sending the c vectors (which are the same size as the model) to each client and back. That doubles communication cost per round. In scenarios with few clients, this overhead is manageable, but note it in your analysis as a con.

If the implementation is too complex, as an alternative you can conceptually discuss how you would implement SCAFFOLD. But we encourage trying, as it's good practice with state management in PyTorch. Use smaller models if needed to debug.

Task 4.3: Gradient Harmonization (FedGH) Gradient Harmonization (FedGH) is a more recent technique (from 2023 research) that addresses client drift by directly modifying the set of gradients at the server before aggregation. The idea is to detect when clients' updates are pointing in opposite directions (which indicates their objectives are in conflict, likely due to non-IID data). If two gradients are in conflict (their cosine similarity is negative), FedGH "harmonizes" them by removing the conflicting components, essentially making them more orthogonal. By doing so for all pairs, the resulting gradients can be averaged without canceling each other out.

How to implement:

- After each client sends its update (gradient or weight delta or the new weights), the server will have M update vectors: $g_1, g_2, \ldots, g_M$. Here g_i can be represented as the difference between client i's new model and the old global model ($\Delta\theta_i = \theta_i^{(K)} - \theta_g^t$), or equivalently the gradient times $-\eta$.
- We need to adjust these vectors before averaging. A straightforward implementation:
  - Compute the flat vectors of updates for each client. (If easier, you can work layer by layer, but conceptually flattening is fine.)
  - For each unordered pair of clients i, j, compute the dot product $g_i \cdot g_j$. Also compute their magnitudes $\|g_i\|$ and $\|g_j\|$.
  - If $g_i \cdot g_j < 0$ (negative inner product, i.e., angle $> 90°$), then we have a conflict. Harmonize by projecting each onto the orthogonal complement of the other:

    $$g_i' = g_i - \frac{(g_i \cdot g_j)}{\|g_j\|^2} g_j$$

    This subtracts out the component of g_i that is against g_j.

    $$g_j := g_j - \frac{(g_i \cdot g_j)}{\|g_i\|^2} g_i$$

    (Note: when implementing, be careful to use the original values; probably compute a projection amount and apply symmetrically. Alternatively, the FedGH paper suggests doing the projection in a certain sequence; you could simplify by just doing one-way projection, but the cited algorithm does both.)
  - This operation makes g_i and g_j orthogonal (their dot product becomes zero). The conflict is "resolved" since they no longer directly oppose each other.
  - One caveat: doing this for all pairs sequentially, the order might matter. The FedGH paper describes projecting in random order or simultaneously. To keep it simple, iterate through pairs once in some order.
- After harmonizing, average the modified g_i' vectors as usual (weighted by N_i/N if using different sizes).
- Proceed to update the global model with this averaged gradient.

The pairwise check is $\mathcal{O}(M^2)$ in the number of clients. For $M = 5$ or 10, that's fine (e.g., 10 clients $\Rightarrow$ 45 pairs). But note if M were large (like 100), this becomes heavy. FedGH is mostly aimed at cross-silo or small cross-device scenarios due to this overhead. There are potential ways to optimize (clustering gradients, etc.) but not needed for our scale.

Testing FedGH:

- Use a highly non-IID scenario (like the worst-case label split: each client has exclusive classes, which Dirichlet $\alpha \to 0$ approximates).
- Compare FedAvg vs FedAvg+GH (i.e., applying gradient harmonization at the server). Measure the global accuracy over rounds.

- You should see FedGH yields an improved accuracy, especially later in training, as it prevents some oscillation. The FedGH paper reported notable boosts, e.g., +5% absolute in some cases over FedAvg.
- Also observe the convergence speed: sometimes FedGH can reach a given accuracy in fewer rounds because it is effectively doing a more directed update by removing contradictory components.

Make sure to reuse your FedAvg implementation and just add this step on the server side. You might structure it as a function `harmonize_gradients(list_of_updates)` that returns adjusted updates.

Task 4.4: FedSAM – Sharpness-Aware Minimization in Federated Learning FedSAM (Qu et al., ICML 2022) incorporates the Sharpness-Aware Minimization technique into FL. In normal training, an update step finds a direction that reduces the loss. SAM instead finds a perturbation δ that maximizes the loss (within some norm-bounded neighborhood of the current weights), and then updates the weights to minimize the loss at that worst-case perturbed point. The effect is that the model not only tries to reduce loss but also ensure the neighborhood around it has low loss, leading to a flatter minimum which usually generalizes better. In FL, this can help because each client's data is limited and can overfit locally; SAM tries to prevent overly sharp local minima that would not generalize to other clients' data.

Implementation:

Integrating SAM into local training means each batch update on the client is replaced by two computations:

- **Ascent step (find perturbation):** For the current weights w and batch data, compute gradients $\nabla L(w)$. Normalize this gradient and take a small step in that direction:

$$w_{\text{adv}} = w + \rho \frac{\nabla L(w)}{\|\nabla L(w)\|},$$

  (if using ℓ_2 norm, or elementwise for ℓ_∞ norm). The hyperparameter ρ controls the radius of the neighborhood (often a fraction of the learning rate or a fixed small value like 0.05).
- **Descent step:** Compute the gradient of the loss at the perturbed weights w_{adv}: $\nabla L(w_{\text{adv}})$. Then update the original weights by descending on that:

$$w \leftarrow w - \eta \nabla L(w_{\text{adv}}).$$

In practice: You can implement SAM by writing a custom training loop on each client. For each batch:

- Do a forward/backward pass to get gradients at current weights. Use these to modify the weights (or a copy of them) to the perturbed weights.
- Compute the loss/gradients at the perturbed weights. Then restore the original weights and apply the descent step gradient to them.

There are some subtle details: It's important to only perturb weights for the gradient computation, but not permanently. Libraries or existing SAM implementations might help (there are open-source implementations of SAM in PyTorch you can refer to, since doing it manually every step is a bit involved). Use an appropriate ρ. Possibly set $\rho = 0.5\eta$ or something to start (the SAM paper often uses a fixed ρ; check their recommendations). SAM doubles the computation on each batch (two forward-backward passes). For small models and datasets this is okay, but be mindful of the time. If it's too slow, you can try doing SAM every other batch or something as an approximation.

FedSAM vs plain FedAvg: Evaluate FedSAM in a non-IID scenario and compare with FedAvg:

- Look at the final accuracy: FedSAM typically yields a higher global accuracy because the model is better generalized to all data (the paper reported notable improvements across benchmarks).
- It might also reduce the gap between training and test performance on each client – because SAM finds flatter minima that work for more data distributions.
- If possible, also test FedSAM on a nearly IID scenario: it often still can help generalization a bit, but the benefit is more pronounced for heterogeneous data.

Advanced variants: We mentioned FedGMT, FedLESAM, FedGAMMA in the class. These are improvements to FedSAM: **FedLESAM (Locally Estimated SAM)** suggests using the difference between successive global models as the perturbation direction instead of the local gradient, thereby aligning perturbations to a more global landscape and saving computation (only one backward pass needed). Implementing FedLESAM is complex and not required; but interestingly it shows researchers are actively improving SAM for FL. **FedGAMMA (Global**

SAM) attempts to apply SAM at the server on the aggregated model, to directly minimize global sharpness, but that requires second-order info from clients. We won't go there, but be aware such ideas exist. For your experiments, plain FedSAM is sufficient.

Summary for Task 3: For each method (FedProx, SCAFFOLD, FedGH, FedSAM), you should:

- Implement the method.
- Run an experiment; you can use a fixed challenging non-IID setting, e.g. $\alpha = 0.1$ or even more skew, to evaluate it relative to FedAvg.
- Gather metrics like final test accuracy, and perhaps the number of rounds needed to reach a certain accuracy.
- Save the learning curves (accuracy vs rounds). It might be helpful to plot all methods on one graph for a given scenario, to visually compare (FedAvg vs FedProx vs SCAFFOLD vs FedGH vs FedSAM).
- Note any differences in training behavior (e.g. did one oscillate less? converge faster? etc.)
- You do not need to exhaustively grid-search hyperparameters, but do mention what settings you used (especially μ for FedProx, and ρ for SAM, etc.). If a method didn't seem to help much, speculate why – perhaps the scenario wasn't heterogeneous enough, or maybe its hyperparameter needed tuning.

You can further have plots or comparative results for at least one non-IID scenario (e.g., test accuracy over rounds for each method on the same plot, or a bar chart of final accuracies). Do brief discussion of each method's pros/cons observed. For example: *"FedProx ($\mu = 0.01$) stabilized training and improved final accuracy from 60% to 64% under $\alpha = 0.1$, but still lower than IID case. SCAFFOLD nearly matched the IID case, reaching 75%, with faster convergence in early rounds – confirming its effectiveness. However, SCAFFOLD required sending the control variate (same size as model) each round to clients, doubling communication. FedGH gave a modest boost (60% → 66%) and seemed to smooth out the training curve. FedSAM ($\rho = 0.05$) improved accuracy to 70% and reduced overfitting on any single client's distribution, at the cost of 2x local computation per round."* — This kind of analysis (with your actual numbers) is what we expect.

Make sure to cite any reference implementation or formula you used (in comments or report) and give credit. Also, if any method was particularly hard to get working, mention the challenges.

Task 5: Synthesis of results & Report Writing Guidelines: Now that you have conducted the experiments, the final task is to synthesize your findings into a coherent analysis and present it in a professional manner.

Deliverables:

- GitHub Repo: containing code (with documentation), any saved models or data subsets, and a README explaining how to run your analysis.
- PDF Report: ICML style, 6-10 pages including figures, addressing all points above. Treat it as a professional paper – clarity, structure, and correctness are key. Guidelines are given below. If you have additional results, you may add them in appendix after 10 pages. Write clearly and succinctly. Use bullet points or subheadings only if necessary.

Remaining instructions for report writing remain same as for previous assignments. However, for this particular assignment you can focus on organizing the results into subsections if needed:

- **FedSGD vs Centralized:** Present the findings from Task 1. You can say, e.g., "We verified in a toy experiment that FedSGD (1 local step) exactly replicated the trajectory of centralized SGD, confirming theoretical equivalence (Figure X shows overlapping loss curves)." Explain any minor deviations if observed.
- **Impact of Data Heterogeneity on FedAvg:** Present the plots from Task 2. Describe the trend: how accuracy degrades with increasing non-IIDness. You might include a figure with accuracy-vs-round curves, or a table of final accuracies for different α. Comment on why this happens (e.g., with high skew, client updates are often in conflicting directions causing the averaged model to oscillate or converge to a less optimal point).
- **Comparison of Methods (FedProx, SCAFFOLD, etc.):** You could have one subsection per method or a combined subsection where you compare all on one plot. For clarity, a combined comparison graph (all methods in one plot under a high heterogeneity setting) is great to illustrate differences. Additionally, you might have separate small plots highlighting one method at a time if needed. Report the key numbers: e.g., "At $\alpha = 0.1$, FedAvg final accuracy = 60%, FedProx = 62%, FedGH = 66%, FedSAM = 70%, SCAFFOLD = 74%," (hypothetical numbers) and discuss. Also mention any notable observations like convergence speed or stability. Did any method overshoot or diverge? Did any have diminishing returns?
- **Analysis of Pros/Cons:** Discuss trade-offs: communication vs computation. E.g., "SCAFFOLD was most effective but doubled communication and required maintaining state per client. FedSAM improved accuracy but each round took roughly twice as long due to the inner ascent step. FedProx was lightweight to implement but provided only a small benefit in our experiments. Gradient harmonization helped without changing client code at all – a pure server-side fix – but its impact was moderate." Use evidence from your results to support these points.
- If you measured anything extra (like the divergence metric or time per round), include that discussion too.

Follow a formal academic writing style. Use clear headings for each section as suggested. Include figures (plots) with captions. When you refer to a figure in text, use something like "Figure 2 shows the accuracy of FedAvg under different Dirichlet α." For math, use LaTeX formatting (as we've done throughout this assignment description). Aim for clarity and brevity – focus on insights from the experiments rather than just re-stating theory.

Before finalizing, proofread your report to ensure it's cohesive and that someone who reads it can understand the main takeaways without having to read this assignment prompt. Also make sure all figures and tables are properly labeled and referenced.

References:

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), 2017. (Introduced Federated Averaging algorithm)

[2] T. Li, A. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated Optimization in Heterogeneous Networks," Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys), 2020. (Proposed FedProx, addressing stability under heterogeneity)

[3] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. Suresh, "SCAFFOLD: Stochastic Controlled Averaging for Federated Learning," Proceedings of the 37th International Conference on Machine Learning (ICML), 2020. (Introduced control variates to correct client drift in FL)

[4] D. A. E. Acar, Y. Zhao, R. M. Navarro, M. Mattina, P. Whatmough, and V. Saligrama, "Federated Learning based on Dynamic Regularization," Proceedings of the International Conference on Learning Representations (ICLR), 2021. (FedDyn algorithm that aligns local and global optima via dynamic regularization)

[5] X. Zhang, W. Sun, and Y. Chen, "Tackling the Non-IID Issue in Heterogeneous Federated Learning by Gradient Harmonization," arXiv preprint arXiv:2309.06692, 2023. (FedGH method to mitigate gradient conflicts in FL)

[6] Z. Qu, H. Tang, S. Wang, and L. Sun, "Generalized Federated Learning via Sharpness-Aware Minimization," Proceedings of the 39th International Conference on Machine Learning (ICML), 2022. (FedSAM algorithm incorporating SAM into federated learning)

[7] Z. Fan, Y. Wu, and T. Li, "Locally Estimated Global Perturbations are Better than Local Perturbations for Federated Sharpness-Aware Minimization," arXiv preprint arXiv:2405.18890, 2024. (FedLESAM – an improved, efficient variant of FedSAM)

[8] R. Dai, J. Zhao, and X. Li, "FedGAMMA: Federated Learning with Global Sharpness-Aware Minimization," IEEE Transactions on Neural Networks and Learning Systems (TNNLS), 2023. (Uses global model perturbations to improve federated training stability)

[9] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated Learning with Non-IID Data," arXiv preprint arXiv:1806.00582, 2018. (Early study on impact of non-IID data for FedAvg, suggesting data-sharing or weighting strategies)

[10] P. Kairouz, H. B. McMahan, B. Avent, et al., "Advances and Open Problems in Federated Learning," Foundations and Trends in Machine Learning, vol. 14, no. 1–2, pp. 1–210, 2021. (Comprehensive survey of FL, discusses challenges like heterogeneity and approaches)