FedAvg

FedAvg Algorithm

1. **Server Initialization:** Server initializes global model w^0 and broadcasts to participants
2. **Local Training:** Each selected client k trains on local data D_k for E epochs using SGD: update w_k^{t+1} (based on w_k^t)
3. **Upload Updates:** Clients send updated weights w_k^{t+1} to server
4. **Aggregation:** Server computes weighted average of client models

$$w^{t+1} \leftarrow \Sigma_k \ (n_k/n) \cdot w_k^{t+1}$$

1. **Broadcast & Repeat:** Server broadcasts w^{t+1} to all clients. Repeat steps 2-5 until convergence

FedAvg - Simple Implementation

- Works with any model: Neural networks, linear models, tree ensembles

Example: 5 hospitals want to train pneumonia detector

Each hospital: 10K chest X-rays, ResNet-18 model

1. Each hospital trains ResNet locally for 5 epochs (the same number of examples)
2. Upload 11M parameters (44MB) to server
3. Server computes: $w\_global = 0.2 \cdot w_1 + 0.2 \cdot w_2 + 0.2 \cdot w_3 + 0.2 \cdot w_4 + 0.2 \cdot w_5$
4. Broadcast updated model back to hospitals
5. Repeat for 100 rounds

Good accuracy without sharing any patient data!

FedAvg - Privacy Preservation

- No data centralization: Server never sees raw images, text, or records
- Regulatory compliance: Easier to satisfy GDPR
- User trust: Customers more willing to participate when data stays local

1. Each client trains locally
2. Each client sends only model: w
3. Server aggregates updates from many devices
4. Improved model sent back to all clients

- Model weights can still leak some information (gradient attacks) -> secure aggregation or differential privacy.

FedAvg - Communication Efficiency

- Local training reduces the need for constant server communication
- Multiple local epochs: Training locally before communicating
- Batch updates: Send weights once per round, not after every gradient step
- Works with slow networks: Tolerates high latency connections

Local computation is cheap, communication is expensive - FedAvg optimizes for this

FedAvg - Scalability

- A large number of clients:
    - Client sampling: Select only 1-10% of clients per round
    - Asynchronous updates: use fast clients

Train on many data without having large infrastructure

FedAvg - Heterogeneous Data Distribution

Slow convergence: more rounds to converge

- Poor global performance: Model good locally but bad globally
- Weight divergence: Client models drift in different directions
- Unfairness: Model performs well for some clients, poorly for others

Example: Handwriting Recognition (MNIST) - 10 clients train digit classifier - Non-IID distribution:

- Client 1: Only digits 0-1
- Client 2: Only digits 2-3
- Client 3: Only digits 4-5
- ...

- Aggregated model: Confused, accuracy drops

FedAvg - Communication Overhead

Large Model Sizes - modern deep learning models are huge - transmitting them is expensive

- Bandwidth bottleneck: Upload speeds much slower than download (asymmetric)
- Battery drain: Mobile devices drain battery during uploads
- Costs money: Cellular data expensive, users may not opt-in
- Slow convergence: Waiting for uploads delays training

Example: Mobile Image Classification - model size: 14MB (3.5M parameters)

- Per round per client: 14MB upload + 14MB download = 28MB

User impact:

- User pays
- Battery
- Time

FedAvg - Straggles Problem

Slow Clients Delay Training - Synchronous aggregation must wait for slowest client

- Device heterogeneity - huge speed difference
- Network variability - WiFi vs 3G vs offline devices
- Waiting time - Fast clients idle while waiting for stragglers
- Dropouts - Some clients never finish, round must be restarted

- Include straggler (slow) or exclude (lose data diversity)?

FedAvg - Privacy Leakage Risks

Model Updates Can Leak Information - gradients/weights can reveal private information

- Gradient inversion: Reconstruct training images from gradients
- Membership inference: Determine if person was in training set
- Property inference: Learn statistical properties of data
- Model inversion: Extract representative samples from model

Example: Gradient Inversion Attack

- Can reconstruct high-quality images from gradients (image classification)
- Privacy guarantee not as strong as it seems

FedAvg - System Heterogenity

Diverse Hardware and Software - clients have vastly different computational capabilities

- Hardware differences: CPU vs GPU
- Software differences: Different ML frameworks, Python versions, OS
- Computation time: differences between fastest and slowest devices

Example: Training on diverse mobile devices

- Cannot use same model size for all devices
- Cannot require same number of epochs
- Weak devices drop out or never participate
- Biased towards data from powerful devices

- Model trained mostly on high-end device data

FedAvg

Benefits

- Simple to implement
- Privacy preserving (no raw data shared)
- Communication efficient
- Highly scalable
- Works with any model architecture

Drawbacks

- Struggles with non-IID data
- Large communication overhead (big models)
- Stragglers slow down training
- Gradient attacks can leak info
- System heterogeneity challenges

When to Use FedAvg?

Good Fit:

- Data is naturally distributed
- Privacy is critical
- Data reasonably balanced

Poor Fit:

- Highly skewed data
- Need fast convergence
- Limited bandwidth
- Very heterogeneous devices

FedProx - Federated Proximal

- Solving Non-IID Data Challenges - handles data heterogeneity and system heterogeneity

FedAvg Problems:

- Assumes IID data
- Client drift: local models diverge
- Slow convergence with heterogeneity
- Unstable training dynamics

FedProx Solution

- Add proximal term to loss
- Prevents excessive drift
- Faster, more stable convergence

$$\text{Loss} = \text{Original\_Loss} + (\mu/2)||w - w_{global}||^2$$

Proximal term: $(\mu/2)||w - wt||^2$ keeps local weights close to global model

FedProx - Client drift

Example: MNIST Digits: 10 clients, each gets mostly one digit

Client distributions:
- Client 1: 90% digit 0, 10% others
- Client 2: 90% digit 1, 10% others
- ...
- Client 10: 90% digit 9, 10% others

Round 1: Global model accuracy = 50%
- Client 1 trains locally: great at digit 0 (95%), bad at others (20%)
- Client 2 trains locally: great at digit 1 (95%), bad at others (20%)

Round 2: After aggregation, global model = 55%

- Each client pulls in opposite direction
- Averaging cancels out improvements
- For convergence - many rounds

FedProx - Client drift

- Non-IID data - clients optimize toward different objectives
- Client drift: Local model parameters move away from global model in conflicting directions

• Client A's data: Mostly class 0 → optimizes for class 0

• Client B's data: Mostly class 1 → optimizes for class 1

• Global goal: Perform well on both classes

Result: Local updates conflict when aggregated!

Client drift happens because local objectives diverge from global objective

The Proximal Term

The penalty keeps local model close to global model

$$\text{Loss} = \text{Original\_Loss} + (\mu/2)||w - w_{global}||^2$$

Similar a rubber band:

- Global model w^t: Fixed anchor point
- Local model w: Can move to fit local data
- Proximal term: Rubber band connecting them
- μ: Stiffness of rubber band

During training:

- Original loss pulls w toward local optimum
- Proximal term pulls w back toward global model
- Final w is a compromise between the two
- Prevents w from drifting too far away

Local model improves on local data while staying connected to global knowledge

FedProx

Client k (Local Training)

ClientUpdate(k, w_t):

$w \leftarrow w_t$ (initialize with global model)

for epoch e = 1 to E:

for batch B in local data D_k:

- Compute loss: $L = (1/|B|) \Sigma_{(x,y)\in B} \ell(w; x, y)$
- **Add proximal term:** $L \leftarrow L + (\mu/2)||w - w_t||^2$
- Compute gradient: $g \leftarrow \nabla L$
- Update: $w \leftarrow w - \eta \cdot g$

return w

Server (Coordinator)

Initialize: w^0

for each round t = 0, 1, 2, ..., T:

1. **for** each client k:
   - Send w_t to client k
   - $w_{kt+1} \leftarrow$ ClientUpdate(k, w_t)
2. Aggregate: $w_{t+1} \leftarrow \Sigma_{k\in S_t} (n_k/n) \cdot w_{kt+1}$

FedAvg:

$L = (1/|B|) \Sigma \ell(w; x, y)$

FedProx:

$L = (1/|B|) \Sigma \ell(w; x, y) + (\mu/2)||w - wt||^2$

Computational Overhead

- Extra computation: Computing $||w - w_t||^2$ once per batch
- Overhead: ~1-5% extra time
- Memory: Store w_t (already needed anyway)

FedProx - Effect of Different μ Values

μ = 0 (FedAvg)

- Local models free to drift
- Fast local convergence
- Slow global convergence (500 rounds)
- Final accuracy: 75% (poor with non-IID)

μ = 0.001 (Weak)

- Slightly reduces drift
- Convergence: 400 rounds
- Final accuracy: 78%
- Minimal improvement

μ = 0.01 (Recommended)

- Good balance local/global
- Convergence: 200 rounds
- Final accuracy: 85%

μ = 0.1 (Strong)

- Strong regularization
- Convergence: 150 rounds
- Final accuracy: 83%
- Good for extreme non-IID

μ = 1.0 (Too Strong)

- Over-regularized
- Convergence: 300 rounds
- Final accuracy: 80%
- Local models cannot adapt enough

FedProx - Effect of Different μ Values

What Does μ Control - the trade-off between local and global optimization

- μ = 0: No regularization → Standard FedAvg
- μ → ∞: Infinite regularization → w cannot move at all
- μ small (0.001-0.01): Gentle nudge toward global
- **μ medium (0.01-0.1): Balanced trade-off**
- μ large (0.1-1.0): Strong constraint
- μ very large (>1.0): Slow learning

Example

Training pneumonia detector across 5 hospitals with different patient demographics

5 Hospitals with Non-IID Data:

- Hospital A: 10K X-rays, avg age 75, 60% pneumonia
- Hospital B: 8K X-rays, avg age 8, 20% pneumonia
- Hospital C: 12K X-rays, avg age 45, 30% pneumonia
- Hospital D: 5K X-rays, avg age 60, 80% pneumonia
- Hospital E: 3K X-rays, avg age 50, 25% pneumonia

Model: ResNet-18

- 11M parameters
- Input: 224×224 grayscale X-ray
- Output: Binary (pneumonia / healthy)

Example

Baseline: FedAvg (μ=0)

Results after 200 rounds:

- Hospital A: 85% accuracy (good on elderly, bad on others)
- Hospital B: 70% accuracy (good on children, bad on adults)
- Hospital C: 78% accuracy (mediocre on all)
- Hospital D: 82% accuracy (good on severe, bad on mild)
- Hospital E: 65% accuracy (limited data, poor learning)
- **Global test set: 72% accuracy**

Models specialized to local data, poor global performance

Solution: FedProx (μ=0.01)

Results after 100 rounds (2x faster!):

- Hospital A: 88% accuracy
- Hospital B: 86% accuracy
- Hospital C: 87% accuracy
- Hospital D: 89% accuracy
- Hospital E: 84% accuracy
- **Global test set: 87% accuracy**

Improvement: 72% → 87%

FedProx ~ Stay near the global model while training: limits movement

Scaffold - Stochastic Controlled Averaging for Federated Learning

Example: Binary Classification ($y \in \{0, 1\}$) Model: Linear classifier $w = [w1, w2]$ and

3 Clients with heterogeneous data:

- Client 1: Only class 0 samples (only cats)
- Client 2: Only class 1 samples (only dogs)
- Client 3: Balanced (50% cats, 50% dogs)

Scaffold: If you drift, I'll guide your gradient back toward the global direction: give information to stay oriented

Scaffold

Round 1 - FedAvg:

Server: sends w_0 = [0, 0] to all clients

Client 1 (only class 0): // Gradient always points toward class 0 -> Local training w → [-2, -2]

Client 2 (only class 1): // Gradient always points toward class 1 -> Local training w → [+2, +2]

Client 3 (balanced): // Gradients balanced Local training w → [0, 0]

Server aggregation: w_1 = avg([-2,-2], [+2,+2], [0,0]) = [0, 0]

Round 2 - FedAvg:

Server: sends w_1 = [0, 0] to all clients

Client 1: w → [-2, -2]

Client 2: w → [+2, +2]

Client 3: w -> [0, 0]

Server: w_2 = [0, 0]

- Clients keep drifting in opposite directions
- Each round, clients waste computation moving in directions that cancel out.

Scaffold

Client Drift

- **Slow convergence**: Updates pull the model in different directions
- **Poor final accuracy**: Model may converge to suboptimal solutions
- **Instability**: Training can diverge in extreme heterogeneity

- **Control variates** to correct for the bias introduced by heterogeneous client data.
- Maintains control variates (correction terms) at both the server and client levels to track and compensate for the drift between local and global objectives.

Control variates - variance reduction:

- **Server control variate (c)**: Tracks the average drift across all clients
- **Client control variates (c_i)**: Track individual client drift
- **Correction term**: (c_i - c) compensates for local bias

Scaffold - Control Variates

Use **control variates** at client and server to **correct local gradients**, ensuring clients move in roughly the same direction.

Corrected Estimate = Biased Estimate - (Control Variate - Expected Control)

SCAFFOLD applied to gradient updates:

Corrected gradient = $g_i - (c_i - c)$

- g_i = local gradient (biased)
- c_i = client control variate (tracks local drift)
- c = server control variate (tracks average drift)

The correction term $(c_i - c)$ is a bias correction - aligns each client's update with the global average direction.

Scaffold

Initialization: $c_1 = [0, 0]$, $c_2 = [0, 0]$, $c_3 = [0, 0]$ $c = [0, 0]$

Round 1 - SCAFFOLD:

Client 1 (only class 0):

True gradient: $\nabla f1(w) = [-1, -1]$

Corrected gradient: $\nabla f1(w) - (c_1 - c) = [-1, -1] - [0,0] + [0,0] = [-1, -1]$

After local training: w_new = [-2, -2]

Update control variate:

Update c_1_new = [-1, -1] // Captures the drift direction!

Client 2 (only class 1): …

c_2_new = [+1, +1] // Captures opposite drift!

Client 3 (balanced):

c_3_new = [0, 0]

Server:

Updates global $c = avg(c_1, c_2, c_3) = [0, 0]$

Scaffold

Round 2 - SCAFFOLD (THE KEY DIFFERENCE):

Client 1:

True gradient: $\nabla f1(w) = [-1, -1]$

Corrected gradient: [-1,-1] - [-1,-1] + [0,0] = [0, 0] // The control variate CANCELS the systematic drift!

Client 2:

True gradient: $\nabla f2(w) = [+1, +1]$

Corrected gradient: [+1,+1] - [+1,+1] + [0,0] = [0, 0] // Drift cancelled here too!

…

Result: Clients now make progress toward global optimum instead of drifting in opposite directions!

Scaffold

FedAvg after 100 rounds:

Client 1 local model: [-2.0, -2.0] (heavily biased toward class 0)
Client 2 local model: [+2.0, +2.0] (heavily biased toward class 1)
Global model: [0, 0] (correct but via cancellation)
Test accuracy: 75% (poor generalization)

SCAFFOLD after 100 rounds:

Client 1 local model: [0.1, -0.1] (close to global optimum)
Client 2 local model: [-0.1, 0.1] (close to global optimum)
Global model: [0, 0]
Test accuracy: 95% (better)

- FedAvg: Clients keep pulling in opposite directions
- SCAFFOLD: Control variates remember and correct for systematic bias

Scaffold

Server

Sends the current global model w^t and its control variate c^t to the selected clients.

Receives updates $(\Delta w_k^t, \Delta c_k^t)$ from each client from S_t

- Aggregate model updates (similar to FedAvg):

$$\Delta w^t = \frac{1}{|S_t|} \sum_{k \in S_t} \Delta w_k^t$$

- Update the global model:

$$w^{t+1} = w^t + \eta_g \Delta w^t$$

(η_g is the server learning rate, often set to 1).

- Aggregate control variate updates:

$$\Delta c^t = \frac{1}{|S_t|} \sum_{k \in S_t} \Delta c_k^t$$

- Update the server control variate:

$$c^{t+1} = c^t + \Delta c^t$$

Scaffold

Client (k)

- Initializes its local model $w_k^t = w^t$.
- Receives the server control variate c^t.
- Performs E local steps of stochastic gradient descent (SGD). For each local step $\tau = 0, ..., E-1$, using a local minibatch b:
  - Compute the local gradient: $g_k(w_{k,\tau}^t) = \nabla F_k(w_{k,\tau}^t; b)$, where F_k is the local loss function for client k.
  - Update the local model using the corrected gradient:

    $$w_{k,\tau+1}^t = w_{k,\tau}^t - \eta_l(g_k(w_{k,\tau}^t) - c_k^t + c^t)$$

    Here, η_l is the local learning rate. Notice how the local gradient g_k is adjusted by the difference between the server control variate c^t and the client's control variate c_k^t. This adjustment aims to correct for the client's local drift.
- Compute the total model update direction: $\Delta w_k^t = w_{k,E}^t - w^t$.
- Update the client control variate c_k. A common way is:

  $$c_k^{t+1} = c_k^t - c^t + \frac{1}{E\eta_l}(w^t - w_{k,E}^t)$$

  This update reflects the average local gradient direction observed during the E steps.
- Compute the change in the client control variate: $\Delta c_k^t = c_k^{t+1} - c_k^t$.
- Send Δw_k^t and Δc_k^t back to the server.

Scaffold

SCAFFOLD = FedAvg + Smart Drift Correction

SCAFFOLD Solves Client Drift

- Uses control variates to correct biased local gradients
- Achieves unbiased updates even with non-IID data
- Provably faster convergence than FedAvg

When to Use SCAFFOLD

- **High data heterogeneity**: When clients have very different data distributions
- **Partial participation**: When only sampling subset of clients per round
- **Need for convergence guarantees**: When theoretical guarantees matter

Practical Considerations

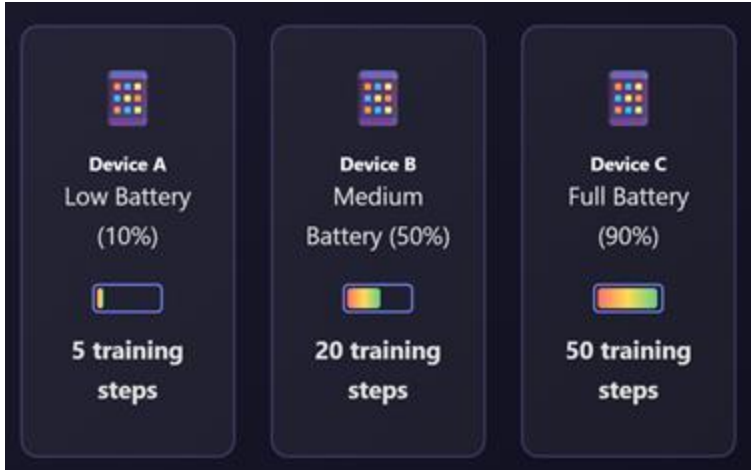
- **Memory**: Requires storing control variates (same size as model)
- **Computation**: Minimal overhead (one extra vector addition per gradient)
- **Communication**: Same as FedAvg (no additional rounds needed)

FedNova - Federated Normalized Averaging

- Handling heterogeneous local computational capabilities across different devices
- Mobile devices have varying battery levels -> perform different numbers of local training steps

Standard FedAvg treats all updates equally, regardless of the number of local steps performed

- Devices with low battery might only do a few gradient steps
- Devices with full battery can perform many more steps
- Some devices might have better processors and can compute faster



FedNova

The Bias Problem

- Devices with more computational resources - dominate the global model
- Low-battery devices' contributions are overshadowed
- Model convergence becomes suboptimal and unfair

Real-World Impact

- Mobile devices have varying battery levels throughout the day
- Different hardware capabilities (CPU, RAM) across device models
- Network conditions affect how long devices can train
- User behavior impacts available computation time

FedNova

FedNova normalizes the local updates by the actual number of steps performed by each device:

1. **Local Training**: Each device i performs K_i local steps (K_i varies by device based on battery, computational power, etc.)
2. **Normalized Update** - depends on the client's number of local steps and learning rate: remove the bias caused by some clients training longer than others
3. **Server Aggregation**: normalized updates are sent to the server
    - The server collects all normalized updates and combines them using data-size weighting

FedNova

Fairness

Every device contributes proportionally to its actual computation, not just participation

Better Convergence

Removes bias from heterogeneous training steps, leading to optimal model performance

Battery-Aware

Low-battery devices can still contribute meaningfully without draining resources

Flexibility

Each device adapts computation to current conditions (battery, CPU load, etc.)

FedNova

Client-Side Requirements

- Track the number of local training steps performed
- Compute normalized update before transmission
- Report step count to server along with update

Server-Side Requirements

- Collect step counts from all participating devices
- Apply weighted aggregation using step counts
- Maintain global learning rate schedule

Communication Overhead

- Minimal: Only one additional scalar (step count) per device
- Same model update size as FedAvg

Performances - Empirical Findings

Convergence Speed

- Up to **2x faster** convergence in highly heterogeneous settings
- More stable training with reduced oscillations

Model Accuracy

- **5-10% improvement** in final model accuracy
- Better generalization across diverse device populations

Fairness Metrics

- Variance in device contributions reduced by **80%**

FedNova

- Solves the fundamental bias problem in heterogeneous federated learning
- Enables fair participation from all devices regardless of capabilities
- Improves both convergence speed and model quality
- Minimal implementation overhead with significant benefits
- Less effective than SCAFFOLD for drift

FedOpt - Adaptive Optimization to Federated Learning

- FedAvg uses simple averaging → global step size fixed
- Server aggregation as an optimization step
- Use adaptive optimizers: Adam, Adagrad, etc on the server

- Faster convergence
- Server has computational resources
- Data is highly non-IID
- Communication rounds are expensive

Comparisons

Comparison Table

Aspect	FedAvg	FedProx	SCAFFOLD	FedNova	FedOpt
Year	2017	2018	2020	2020	2021
Core Innovation	Averaging local models	Proximal term regularization	Control variates for drift correction	Normalized averaging	Server-side optimization
Problem Addressed	Basic FL	System & data heterogeneity	Client drift	Objective inconsistency	Slow convergence
Key Mechanism	Simple averaging	μ-FedProx penalty	Variance reduction	Normalized aggregation	Server momentum/adaptation
Computation Overhead	Baseline	Minimal (+1 term)	High (2x gradients)	Minimal	Minimal
Communication Overhead	Baseline	Same as FedAvg	2x (control variates)	Same as FedAvg	Same as FedAvg
Convergence Rate	$O(1/T)$	Better than FedAvg	$O(1/T^2)$ - Linear	Better than FedAvg	Better than FedAvg

Comparison

FedProx:

- Simple proximal term
- Good convergence (not quite as fast as SCAFFOLD)
- Minimal overhead
- No extra communication

SCAFFOLD:

- Uses control variates to correct drift
- Faster convergence than FedProx (typically)
- More complex: requires storing control variates
- Higher communication cost (send control variates)

FedNova:

- Normalizes by number of local steps
- Solves varying local epochs problem
- Doesn't directly address non-IID

- **Start with:** FedAvg (baseline)
- **If non-IID data:** Try FedProx (μ=0.01)
- **If very non-IID:** Use SCAFFOLD
- **If slow convergence:** Add FedOpt
- **If heterogeneous systems:** Use FedNova

Performance in Federated Learning

Algorithmic Factors

- Aggregation method
- Model architecture

Data related factors

- Data heterogeneity (non-IID data)
- Data quality and quantity
- Class distribution skew

Communication Factors

- Communication efficiency
- Communication rounds

Client-Related Factors

- Client participation and availability
- Computational heterogeneity

System factors

- Privacy and security mechanisms
- Client selection strategy

Communication Solutions

Communication Solutions

Quantization: From 32-bit to 8-bit

- Float32: 01000001 00111001 10011001 10011010 (4 bytes)
- Int8: 01001110 (1 byte)

Benefits

- Bandwidth reduction (32 bits -> 8 bits per parameter)
- Faster computation (integer ops faster than floating point)
- Memory efficient: 4x less RAM required
- Energy savings: less data to transmit

Trade-offs

- Small accuracy loss
- Quantization noise can accumulate over rounds

Communication Solutions

Top-K Sparsification

Top-K Selection: Select only K largest gradients by absolute value

$$\text{Sparse}(\nabla W) = \{\nabla w_i : |\nabla w_i| \in \text{top-K}(|\nabla W|)\}$$

Communication Solutions - Top-K Sparsification

After local training, each client:

1. Computes gradients or model updates
2. Selects the Top-K elements with the largest absolute values
3. Sends only these K values along with their indices to the server
4. Discards or accumulates the remaining smaller gradients locally

Gradient Accumulation (Error Feedback)

- The gradients that weren't selected in the current round aren't lost
- They are accumulated locally and added to the gradients in the next round
- Consistent gradients will be transmitted.

Communication Solutions - Top-K Sparsification

Server:

Client A might send updates for parameters {1, 5, 100, 203...}

Client B sends updates for parameters {2, 7, 99, 205…}

1. The server maintains a global parameter vector
2. For each received sparse update from client i:
    - Extract the K indices and their corresponding values
    - Add these values to the corresponding positions in the global aggregation buffer
    - Keep track of how many clients contributed to each parameter position
3. After receiving all client updates, compute the average:
    - For each parameter position, divide by the number of clients that contributed to it
    - Parameters that no client updated remain unchanged or use the previous global value

Communication Solutions ResNet50 - Case Study

Example: compression results: ResNet50

Method	**Size**	**Compression**	**Accuracy**
Original (FP32)	98 MB	1x	76.1%
Quantized (INT8)	24.5 MB	4x	75.8%
Top-1% Sparse	0.98 MB	100x	75.2%
Combined	**245 KB**	**400x**	**74.9%**

Communication Solutions

Quantization

Moderate compression needed (2-4x)

Hardware supports int8 ops

Limited bandwidth: Mobile, IoT devices

Large models: 10M+ parameters

High R: Frequent communication

Simple implementation required

Top-K Sparse

Extreme compression needed (10-100x)

Network is severely limited

Can tolerate small accuracy loss

Have compute for sorting

Communication Solutions ResNet50 - Case Study

Local gradients computed:

- Parameter 1: gradient = -0.001
- Parameter 2: gradient = 0.0005
- Parameter 3: gradient = -0.0003
- Parameter 4: gradient = 0.5
- Parameter 5: gradient = -0.4

Top-2 sparsification:

- Sent: Parameters 4, 5
- Dropped: Parameters 1, 2, 3

- Small gradients, when accumulated over many training iterations, can lead to significant parameter updates
- Dropping small gradients repeatedly
  - Parameters with consistently small gradients never get updated
  - The model fails to fine-tune details

Communication Solutions - Compensation Techniques

Error Feedback

residual = zeros(N_params)

At each training round on client:

1. Compute gradients: ***g = ∇L(model, local_data)***
2. Add previous residuals: ***g_compensated = g + residual***
3. Apply Top-K: ***sparse_g, indices = TopK(g_compensated)***
4. Compute what was dropped: ***residual = g_compensated - sparse_g***
5. Send ***sparse_g*** to server
6. Store residual for next round

- The dropped gradients aren't lost forever—they're accumulated in a "residual" or "error" buffer and added back in the next round.
- Small overhead: only store error locally
- Proven to maintain convergence guarantees

Communication Rounds vs Local Epochs

- **Communication rounds (R)** - number of times clients synchronize their model updates with the central server during training
  - Higher R: more frequent aggregation
  - Increased communication overhead
  - Higher bandwidth consumption
  - More rounds generally: faster convergence but at the cost of communication efficiency

- **Local Epochs (E)** - number of complete passes through local data each client performs before communicating with the server
  - Higher E reduces communication frequency
  - More local computation per round
  - Risk of client drift on non-IID data
  - Improved bandwidth efficiency
  - More local epochs reduce communication but may cause divergence if data is highly heterogeneous

Communication Rounds vs Local Epochs

Total Training Cost

$$\text{Total Cost} = \text{Communication Cost} + \text{Computation Cost}$$

Communication

$$C_{comm} = R \times B \times n$$

R: rounds, B: bandwidth, n: clients

Computation

$$\text{Computation cost} = R \times E \times |D| \times n$$

E: local epochs, |D|: dataset size

Minimize total cost while achieving target accuracy.

Bandwidth-Constrained Scenarios

Communication is the bottleneck - the trade-off

Mobile Networks

Limited bandwidth, variable connectivity, high latency

Healthcare IoT

Distributed sensors with limited transmission capacity

- Communication latency dominates training time:
  - Increasing local epochs is essential to reduce rounds
  - **Low E, High R:** Fast convergence, high communication cost
  - **High E, Low R:** Slower convergence, lower communication cost

The Coordination Challenge

Synchronous: Wait for all clients before aggregating

Asynchronous: Aggregate updates as they arrive

Synchronous: The server waits for all selected clients to complete training and submit updates before performing aggregation and moving to the next round:

1. Server broadcasts global model to selected clients
2. All clients train locally on their data
3. Server waits until ALL clients send updates
4. Server aggregates and updates global model

Asynchronous: The server immediately incorporates updates as they arrive from clients, without waiting for all clients to complete training:

1. Client requests current global model from server
2. Client trains locally and sends update
3. Server applies update immediately (may be stale)
4. Process repeats continuously for each client

Synchronous FL: Characteristics

- Idle server time
- Increased latency
- Unused computation from fast clients

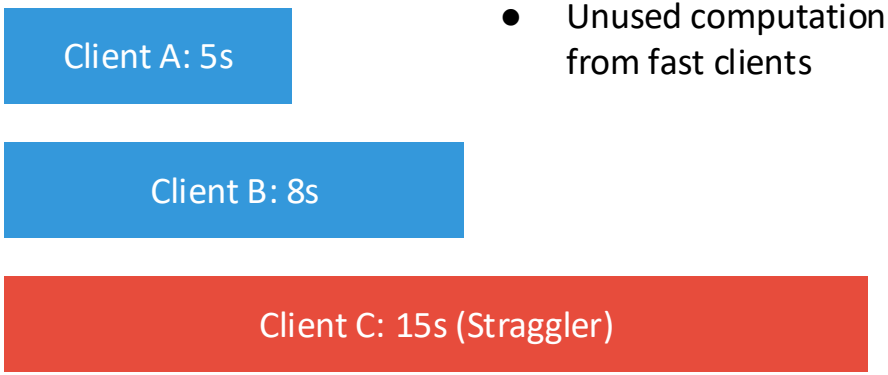
Example: 3 clients training on smartphones with different speeds

Client A (Fast)
Completes in 5 seconds

Client B (Medium)
Completes in 8 seconds

Client C (Straggler)
Completes in 15 seconds

Total round time: 15 seconds



Aggregation -> Round duration 15 sec
Slowest client ("straggler") delays each round → total time increases.

Real-World Challenges

- Devices have different computational speeds
- Network conditions vary dramatically
- Devices go offline unpredictably
- Battery levels affect participation

Synchronous vs Asynchronous FL

Advantages

Synchronous

- **Deterministic convergence:** Proven theoretical guarantees
- **Consistent updates:** All clients on same model version
- **Easy to implement:** Simpler coordination logic
- **Predictable behavior:** Clear round boundaries

Asynchronous

- **No straggler problem:** Fast clients not blocked
- **Better resource utilization:** Continuous updates
- **Fault tolerant:** Single client failure doesn't halt training
- **Higher throughput:** More updates per time unit

Synchronous vs Asynchronous FL

Disadvantages

Synchronous

- **Straggler problem:** Slowest client determines speed
- **Idle time:** Fast clients wait for slow ones
- **Wasted resources:** Server idle during training
- **Poor fault tolerance:** One failure blocks progress

Asynchronous

Staleness: Updates may be based on old models

Convergence issues: Potential for divergence

Complex implementation: Harder to debug

Version management: Track model versions

Asynchronous FL

Example

Slow client downloads model v10, trains for 5 rounds while server advances to v15, then submits stale update.
Staleness: 5 versions behind

Impact

- Can slow or prevent convergence
- May cause model divergence

Balance speed gains against staleness-induced convergence degradation

- The central server periodically **aggregates model updates** sent by clients
- Differences in **network latency, computation speed, or client availability**, some clients **finish their local training later** than others.
- When updating, the server's **global model has already advanced several rounds**.
- Update is **"stale"** — it was computed from an *old version* of the global model.

Asynchronous FL

Client Side

- Pull latest model w_t
- Train locally for E epochs
- Send updated $w_i^{(local)}$ to server

Server Side

- Receive update from client i
- Update w_{t+1} based on $w_i^{(local)}$

The Staleness Problem

Staleness: The age difference between a client's model version and the current global model

Version Timeline

v5 → v6 → v7 → ... → v12

Client starts: v5

Client finishes: v12

Staleness = 7

Example

- Client downloads model version 5
- Trains for 10 minutes
- Meanwhile, server updates to version 12
- Client's update is based on stale model
- Staleness = 12 - 5 = 7

Aggregate in mini-batches: semi-synchronous models

FedBuff (Buffered Aggregation)

- Initialize buffer B = ∅, set buffer size K clients

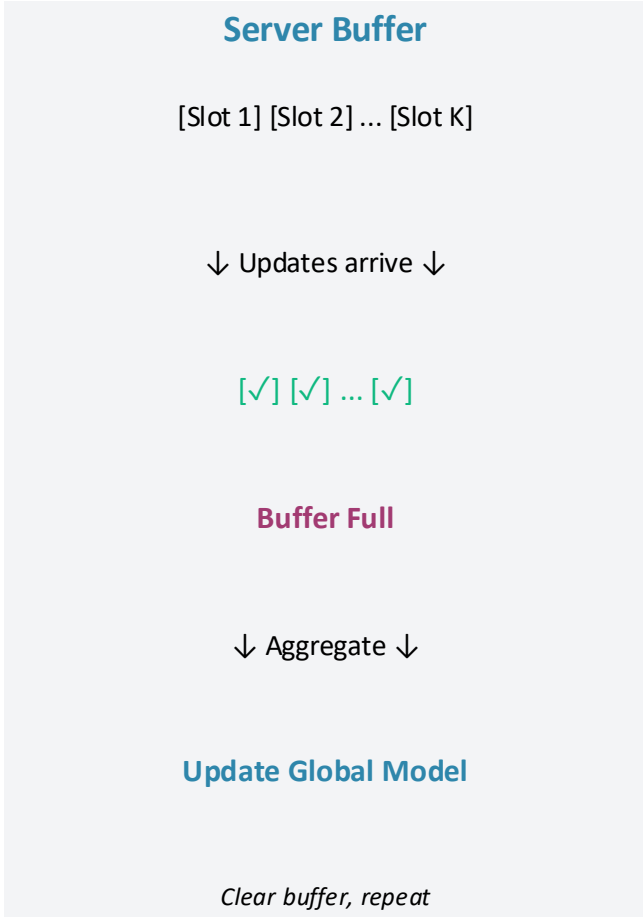
When client i completes: Add Δw to buffer B

- If |B| = K (buffer full):

  Aggregate: w based on clients' gradients

- Clear buffer: B ← ∅
- Broadcast new w to selected clients
- Repeat until convergence

- No strict synchronization required
- Handles stragglers naturally



Asynchronous FedAvg - The Staleness Challenge

Client

- Receive latest global model w_t
- Train locally → produce w_i
- Send back to server.

Server

- initialize w_0
- When client i update arrives:
- Compute staleness $s=t-t_i$
- Apply update with decay factor:

$$w_{t+1} = w_t - \eta \times \text{weight} \times (w_t - w_i)$$

- w_i is the client's model weights

$$\text{weight} = \frac{1}{(1 + \text{staleness})^{\beta}}$$

- weight: staleness penalty - controls how fast the weight decays for stale updates
- β: penalty strength

Comparison Synchronous vs Asynchronous

Aspect	Synchronous	Asynchronous
Coordination	Wait for all clients	Immediate updates
Straggler Impact	High (blocks progress)	Low (no blocking)
Convergence	Deterministic, proven	May require tuning
Fault Tolerance	Poor (one failure blocks)	Excellent (resilient)
Implementation	Simple	Complex (versioning)
Resource Utilization	Low (idle time)	High (continuous)
Best For	Controlled environments	Heterogeneous systems

Semi-Synchronous FL

Example

- Client receives model v5 and starts training
- While training, server updates to v6, v7, v8
- Client finishes and sends update
- Server calculates: staleness = 8 - 5 = 3

Accept updates only if they're not "too stale"

Threshold (τ): Maximum allowed staleness before rejecting an update

Accept if: staleness ≤ τ

Semi-Synchronous FL

Server broadcasts model version vn to selected clients

Clients send updates back asynchronously as they complete

Server checks staleness: if staleness ≤ τ, accept and aggregate

If staleness > τ, reject update and send latest model for retrain

Global model version increments with each accepted update

Adaptive Timeout

Adjust timeout thresholds based on system conditions

- Set initial timeout T
- Monitor client response times
- Adjust T based on percentile statistics
- Accept updates within timeout window
- Balances wait time and participation

Dynamic Timeout

Initial: T = 10s

↓ Monitor ↓

80% respond in 8s

↓ Adjust ↓

New: T = 9s

Balance wait time and participation

When to Use Each Approach

Choose Synchronous FL When:

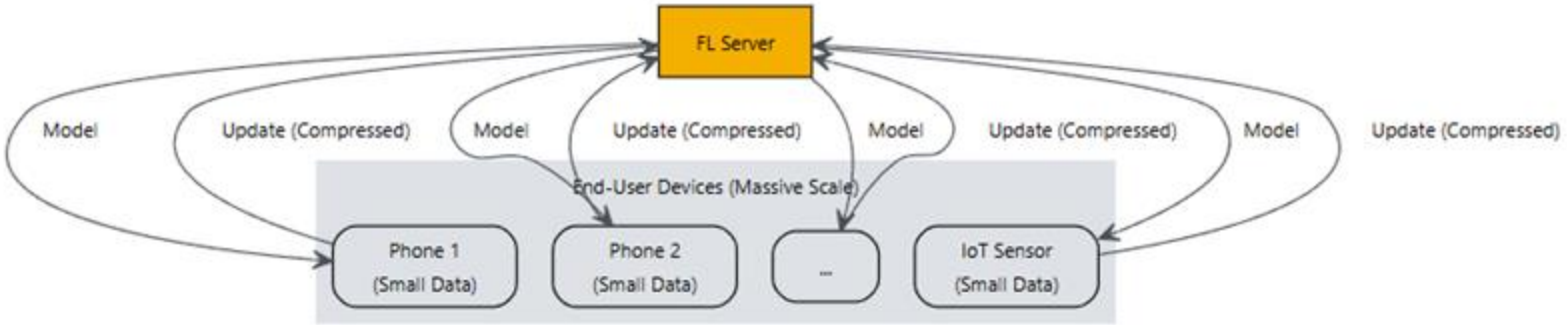
- Similar client capabilities
- Reliable connectivity
- Convergence guarantees critical

Choose Asynchronous FL When:

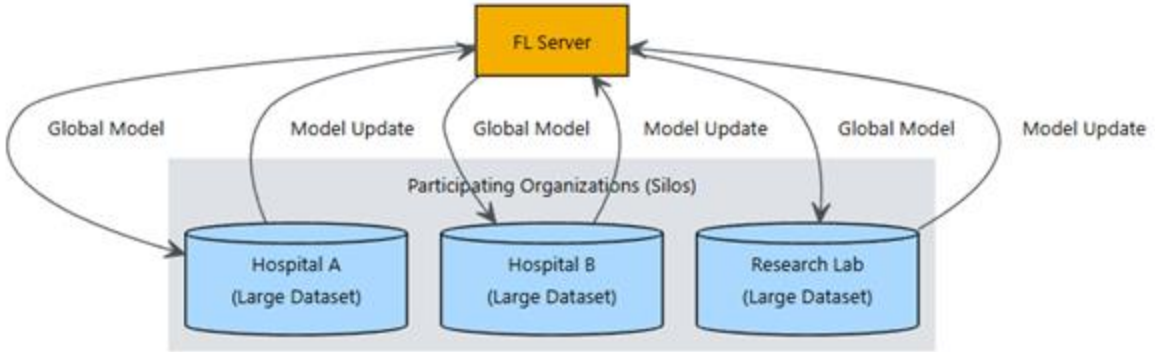
- Heterogeneous speeds (10x+ variance)
- Unreliable connectivity
- Large client populations (1000+)

Cross-device and Cross-silo Federated Learning

Cross-device: very large number of edge devices



Cross-silo: small number of reliable institutions



Cross-device FL and Cross-silo FL

Cross-device FL involves a very large number of edge devices (smartphones, IoT sensors, wearables, etc.) that collaboratively train a global model without sharing their local data.

Each device contributes a small amount of data and only occasionally participates.

Feature	Description
Scale	Millions of devices (e.g., Android phones, IoT nodes)
Data Distribution	Non-IID (each device has its own data pattern) and unbalanced (some have more data than others)
Connectivity	Intermittent — devices may go offline anytime
Participation	Only a random subset participates in each round
Privacy	High concern — data is sensitive (personal usage, text, photos, etc.)
Communication	Bandwidth is limited, so compression and partial updates (e.g., Top-K, quantization) are used
Use Cases	Next-word prediction, keyboard auto-completion, voice recognition, health monitoring

Cross-device FL and Cross-silo FL

Cross-silo FL involves a small number of reliable institutions (called *silos*), each owning large, high-quality datasets — e.g., hospitals, banks, or universities — that cannot share data due to regulations (GDPR, etc.).

Feature	Description
Scale	Dozens to hundreds of silos (not millions)
Data Distribution	May still be non-IID but typically more structured and labeled
Connectivity	Reliable, always-on connections (servers or datacenters)
Participation	Almost all clients (silos) participate in every round
Privacy	Regulatory and business confidentiality (data-sharing forbidden)
Communication	High bandwidth; can afford full model exchanges
Use Cases	Healthcare research, financial fraud detection, cross-bank risk modeling, academic collaboration

Cross-device FL and Cross-silo FL

Example cross-device

Google's Gboard keyboard model:

- Each smartphone locally trains on the user's typing data
- The central server aggregates only the model updates
- After aggregation, an improved model is redistributed to all devices.

Example cross-silo

Several hospitals collaboratively train a medical image diagnosis model:

- Each hospital keeps its patient images locally
- They train on their datasets and share gradients/weights with a central aggregator
- The server aggregates updates and distributes the global model.

Client Selection

Healthcare Federated Learning

- Hospitals have different data sizes and imaging modalities.
  Some hospitals have 1000 MRIs; others only 20.
  Training with all equally causes bias toward small institutions or missing minority data.
  -> client selection ensures balanced representation.

Autonomous-Vehicle Fleet Learning

- Cars in different cities → diverse road/weather data.
- Selecting all would be too expensive; some networks are offline.
  -> client selection contributing novel driving contexts (e.g., snow, rain, night) for faster generalization.

- Client selection - balance **efficiency (less communication)**, **robustness (avoid stragglers)**, and **fairness (represent diverse data)**

Client Selection

1. Server initializes model w_0

2. Select subset of clients S_t

3. Each client trains locally on D_i → sends w_i / Δw_i

4. Server aggregates → new global model w_{t+1}

Which clients should we select each round to optimize training?

- Not all are available (offline, low battery, poor Wi-Fi).
- Some possess redundant or low-quality data.
- Random participation causes slow convergence and unfairness.
- Too many participants ⇒ network congestion, high communication cost.
- **Select a *representative, reliable, and efficient* subset of clients each round.**

Client Selection

N total clients - select K clients per round

Random Selection

- Each round randomly sample K clients

- Pros: simple, unbiased, low overhead
- Cons:
  - May select slow or failing clients
  - Ignores data heterogeneity → slower convergence
  - Unfair: some clients never chosen

Client Selection

N total clients - select K clients per round

Availability-Based Selection

- Only select clients that meet certain availability criteria: client is eligible IF:
  - Device is charging (or battery > threshold)
  - Connected to WiFi (not mobile data)
  - Device is idle (not in active use)
  - Has sufficient storage space
  - Network quality > minimum threshold

- Advantages:
  - Reduces dropout rates
  - Better completion rates
  - Faster round completion
- Disadvantages:
  - Biased towards certain usage patterns
  - May exclude important data distributions
  - Reduces total participating devices

Client Selection

N total clients - select K clients per round

Biased/ Weighted Sampling

- Select clients with probability proportional to some criterion:
  - Data Quantity Weighting
  - Loss-Based Selection: Select clients with higher loss (more room for improvement)
  - Gradient Norm Selection: Select clients with larger gradient updates

Advantages:

- Can prioritize important clients
- Faster convergence in some scenarios
- Adapts to data distribution

Disadvantages:

- Requires knowing client characteristics
- May introduce bias
- Can lead to unfair treatment

Client Selection

N total clients - select K clients per round

Active Selection

- Fairness-Based Selection - Ensure all clients participate fairly over time

```
Initialize: participation_count[i] = 0 for all clients
FOR each round:
        # Compute selection probability inversely proportional to past participation
        FOR each client i:
                p(i) = 1 / (1 + participation_count[i])

        # Normalize probabilities
        p = p / sum(p)

        # Sample based on probabilities
        selected = sample(clients, K, probabilities=p)

        # Update participation counts
        FOR each i IN selected:
                participation_count[i] += 1
```

Client Selection

N total clients - select K clients per round

Active Selection

- Performance-Based Selection - Prioritize reliable, fast clients

Track client metrics:

- Average training time
- Dropout rate
- Update quality
- Network reliability

Score each client:

score(client) = w1 × speed + w2 × reliability + w3 × quality

Select top-K scoring clients

Client Selection

N total clients - select K clients per round

Adaptive Selection

- Performance-Based Selection - Prioritize reliable, fast clients

Early training (rounds 1-50):

```
# Explore - use random selection
strategy = random_selection()
```

Mid training (rounds 51-150):

```
# Focus on high-loss clients
strategy = loss_based_selection()
```

Late training (rounds 151+):

```
# Fine-tune - use diverse selection
strategy = diversity_based_selection()
```

Hierarchical Federated Learning Architecture

Problem	Description
Communication bottleneck	Millions of devices sending updates directly to a central server → high bandwidth load.
Latency & scalability	Global synchronization takes long; stragglers delay training.
Geographical constraints	Devices far from the global server suffer from long transmission delays.
Heterogeneous resources	IoT devices ≠ hospital servers; different compute and communication capabilities.

Hierarchical FL Architecture

Global aggregation, model coordination, final model storage

Regional aggregation, intermediate model updates, client coordination

Local training on private data, gradient computation