FedTinyProp: Adaptive Sparse Backpropagation for Efficient Federated Learning on Embedded Devices

Hatem Trigui
Hahn-Schickard
Villingen-Schwenningen, Germany
Hatem.Trigui@Hahn-Schickard.de

Marcus Rüb
ForestHub
Villingen-Schwenningen, Germany
m.rueb@foresthub.ai

Lilli Frison
Hahn-Schickard
Villingen-Schwenningen, Germany
Lilli.Frison@Hahn-Schickard.de

Axel Sikora
Offenburg University of Applied Sciences
Offenburg, Germany
Axel.Sikora@hs-offenburg.de

Oliver Amft
Hahn-Schickard
Villingen-Schwenningen, Germany
Intelligent Embedded Systems Lab
University of Freiburg
Freiburg, Germany
Oliver.Amft@Hahn-Schickard.de

***Abstract*—We introduce FedTinyProp, a Federated Learning (FL) framework that adapts sparse backpropagation for on-device training. In our approach, clients (edge devices) dynamically sparsify gradients based on per-batch informativeness and skip updates when computation is unnecessary. Clients transmit sparsified weight deltas that are aggregated via a communication-aware Federated Averaging variant. Experiments across FashionMNIST, CIFAR-10/100, UCI HAR, and Google Speech Commands show that FedTinyProp maintains accuracy within 1–6 points of dense baselines while reducing training compute by about 2–5× via adaptive sparsity and skipping. In addition, it reduces client upload volume by up to about 61–86% and lowers peak activation/gradient memory by up to about 45% through sparse backpropagation and sparse delta exchange. Despite the added work of sparse selection, controller overhead is negligible and the savings-to-overhead trade-off remains favorable, with large FLOP savings outweighing non-backpropagation costs.**

I. Introduction

Federated Learning (FL) lets multiple devices collaboratively train a shared machine learning model without centralizing raw data [1], [2]. Moving FL onto resource-constrained embedded and edge devices can be challenging due to tight compute, memory, energy, and uplink constraints [3], [4]. Most deployed algorithms still assume full backpropagation and dense updates, which is costly even for a single local epoch and adds load to low-power or intermittent links [5], [6].

Prior work reduced payloads through compression/quantization or pruned transmitted updates [7]–[9], yet the training step itself often remained dense and expensive for devices. In contrast, sparse backpropagation shows that updating only a subset of parameters can preserve accuracy while lowering training cost [10], [11]. Building on this insight, we adapt principles from TinyPropv2 [12] to the federated setting by making sparsity adaptive to client conditions and by aligning local compute with device capabilities.

This paper provides the following contributions:

1) We introduce FedTinyProp, a batch-wise controller that combines adaptive sparse backpropagation, batch skipping, sparse delta exchange, and sparse-aware aggregation for resource-constrained FL.
2) We evaluate FedTinyProp across multiple benchmarks, including a CIFAR-10 ablation, overhead analysis, and an on-device Raspberry Pi study.

II. Related Work

Sparse backpropagation methods, including meProp and TinyPropv2 reduce training cost by keeping only the largest gradients during the backward pass [10], [12]. Communication-efficient FL reduces the uplink payload by compressing or sparsifying client updates or quantized periodic averaging, including FedPAQ [7]–[9]. The above methods often keep dense local backprop, so client compute and activation/gradient memory remain high. Recent sparse FL methods, including SpaFL and SparsyFed, also introduce sparsity at the model or update level to reduce communication and training cost [13], [14]. FedTinyProp is complementary by focusing on batch-wise sparse backpropagation with skipping to reduce client-side backward compute and activation/gradient memory during local training, while sparse uplink deltas arise naturally as a by-product. Downlink broadcast remains dense and is orthogonal to our approach.

III. Methodology

A. Problem formulation

We minimize the standard FL objective

$$\min_{w} L(w) = \sum_{k=1}^{K} \frac{n_k}{n} L_k(w), \tag{1}$$

where a server coordinates K clients, client k holds n_k samples, $n = \sum_{k=1}^{K} n_k$, and $L_k(w)$ is the local loss at client k. Clients must minimize backward-pass compute, peak activation/gradient memory, and uplink bytes per round.

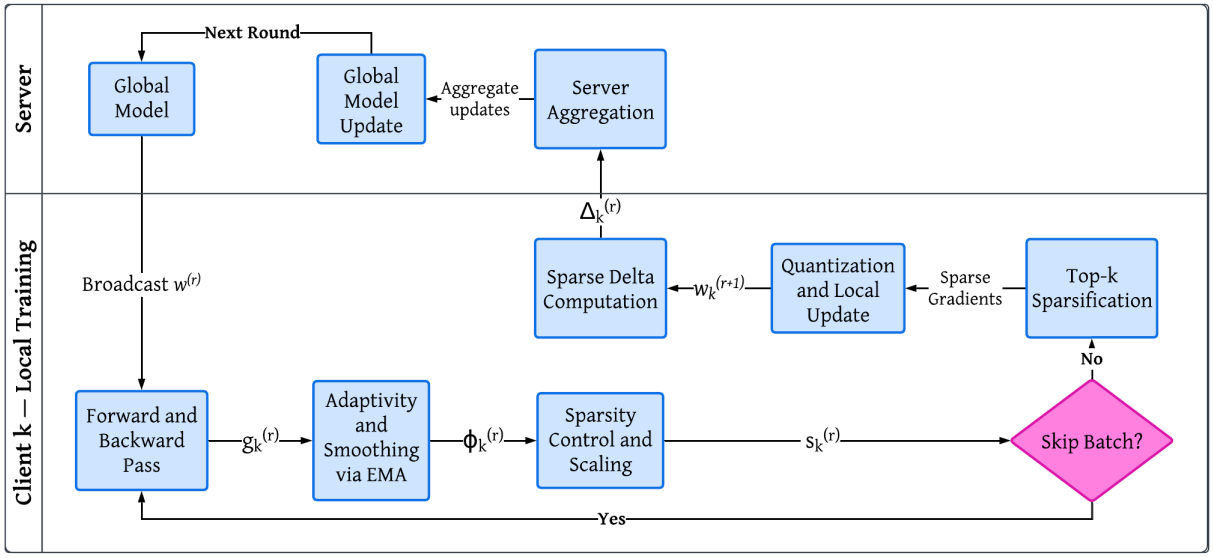

Fig. 1. FedTinyProp pipeline. Each client computes adaptive sparsity and skip decisions per batch before transmitting sparse quantized deltas to the server, which aggregates updates via sparse-aware FedAvg.

B. Summary of the method

FedTinyProp balances FL costs by controlling backward sparsity and the sparsity of transmitted deltas. We focus on sparse backpropagation because it reduces the dominant client costs during training: backward compute and the activation/gradient buffers. Keeping only the Top-k gradient entries per layer gives the best k-sparse approximation in ℓ_2 and yields sparse weight deltas for uplink communication. On the server, sparse updates are aggregated with a sparse-aware FedAvg variant.

An illustrative flow chart of the method is provided in Figure 1. The complete algorithm is provided in Algorithm 1. Default hyperparameters and thresholds used by our implementation are summarized in Table I.

TABLE I
DEFAULT FEDTINYPROP PARAMETERS.

Notation	Name	Description	Default
$S_{\min}$	Min sparsity	Lower sparsity bound	0.5
$S_{\max}$	Max sparsity	Upper sparsity bound	0.95
ζ	Sparsity scale	Maps adaptivity to sparsity	0.5
β	EMA factor	Smoothing for $\tilde{\phi}_k$	0.9
ϕ_{skip}	Skip adaptivity	Skip if $\tilde{\phi}_k$ is below this	0.2
δ_{skip}	Loss threshold	Skip if $\lvert\Delta\ell\rvert$ is below this	0.2
b	Quant. bits	Default quantization bit-width	8
$[b_{\min}, b_{\max}]$	Quant. bit range	Range for adaptive quantization	$[4, 16]$
relative_threshold	Delta rel. thresh.	Delta threshold scale	10^{-3}
minimum_threshold	Delta min thresh.	Min delta threshold scale	10^{-4}

C. Adaptive Sparse Training with Skipping

At client k, for each mini-batch (x, y) we compute an informativeness score from the mean ℓ_2 gradient norm over sparsifiable layers:

$$g_k = \frac{1}{L}\sum_{\ell=1}^{L} \|\nabla_\ell\, \ell(w_k; x, y)\|_2\,, \qquad \phi_k = \frac{g_k}{g_k^{(0)}}, \tag{2}$$

where $g_k^{(0)}$ is a reference from the first few batches. We smooth the informativeness score ϕ_k using an exponential moving average (EMA) to obtain the estimated informativeness $\tilde{\phi}_k$:

$$\tilde{\phi}_k \leftarrow \beta\, \tilde{\phi}_k + (1-\beta)\, \phi_k. \tag{3}$$

We set a batch-wise sparsity level

$$s_k = \text{clip}\Big(\zeta\,(1-\tilde{\phi}_k),\, S_{\min},\, S_{\max}\Big)\,, \tag{4}$$

and optionally skip the backward pass when both the estimated informativeness $\tilde{\phi}_k$ and the loss change $|\Delta\ell|$ are small:

$$\text{skip if } \tilde{\phi}_k < \phi_{\text{skip}} \;\wedge\; |\Delta\ell| < \delta_{\text{skip}}. \tag{5}$$

If not skipped, we apply layer-wise Top-k sparsification with $k = \max(\lfloor N(1-s_k)\rfloor, \lfloor 0.1N\rfloor)$ (where N is the tensor size), update w_k using the sparse gradient, and accumulate sparse weight deltas for uplink. We include a simple fallback that temporarily relaxes sparsity if training becomes unstable, by lowering ζ when a sliding window over recent losses detects large fluctuations.

D. Sparse Delta Communication

After completing local training, each client k computes a sparse update $\Delta_k^{(r)}$ with respect to the received global model $w^{(r)}$. These deltas are determined through an adaptive thresholding mechanism:

$$\Delta_k^{(r)} = \Big\{(i, \delta_i) \;\Big|\; \delta_i = w_{i,k}^{(r)} - w_i^{(r)},\; |\delta_i| > \theta_{\text{adaptive}}\Big\}$$

where $\theta_{\text{adaptive}} = \max(\text{relative\_threshold}, \text{minimum\_threshold})$ and i indexes model parameters, δ_i is the local change for parameter i. These sparse deltas are stored as (index, value) pairs and quantized before transmission.

We quantize only the retained (nonzero) sparse values and transmit them together with their indices. We select the

bit-width b_ℓ per layer from a small discrete set based on local gradient statistics. Per-layer bit-width adaptation reduces uplink bytes compared to a fixed-bit setting while maintaining accuracy.

E. Sparse Server Aggregation

Upon receiving sparse updates $\Delta_k^{(r)}$ from a subset of clients $\mathcal{K}^{(r)}$, the server updates the global model using a sparse-aware variant of FedAvg: unlike standard FedAvg, aggregation only considers parameter indices that were actually updated by at least one client. Let $\mathcal{A}_i = \{k \in \mathcal{K}^{(r)} \mid i \in \text{supp}(\Delta_k^{(r)})\}$ denote the set of clients that updated parameter i. Then the server aggregation follows:

$$w_i^{(r+1)} = w_i^{(r)} + \sum_{k \in \mathcal{A}_i} \frac{n_k}{n} \delta_{i,k}$$

where n_k is the number of local samples on client k, $n = \sum_{k \in \mathcal{K}^{(r)}} n_k$ is the total across all participating clients and $\delta_{i,k}$ is the update for parameter i from client k.

Algorithm 1 FedTinyProp Training Loop (Client & Server)

Initialize global model $w^{(0)}$
for round $r = 0$ to $R-1$ **do**
  Server selects subset $\mathcal{K}^{(r)}$ of clients
  Server broadcasts $w^{(r)}$ to clients in $\mathcal{K}^{(r)}$
  for each client $k \in \mathcal{K}^{(r)}$ **in parallel do**
    Set $w_k^{(r)} \leftarrow w^{(r)}$
    Initialize $\phi_k^{(r)}, \zeta_k^{(r)}, \tilde{\phi}_k^{(r)}$
    for each batch (x_i, y_i) in local data **do**
      Compute predictions $\hat{y}_i$ and loss $\ell^{(r)}$
      Backpropagate gradients via TinyProp layers
      Compute average gradient norm $g_k^{(r)}$
      Update adaptivity $\phi_k^{(r)} = \frac{g_k^{(r)}}{g_k^{(0)}}$
      Smooth via EMA: $\tilde{\phi}_k^{(r)} = \beta \cdot \tilde{\phi}_k^{(r-1)} + (1-\beta) \cdot \phi_k^{(r)}$
      Adapt $\zeta_k^{(r)}$ based on $\phi_k^{(r)}$ and $\Delta\ell$
      Compute batch sparsity $s_k^{(r)} = \texttt{clip}(\zeta_k^{(r)} \cdot (1 - \tilde{\phi}_k^{(r)}), S_{\min}, S_{\max})$
      if $\phi_k^{(r)} < \phi_{\text{skip}}$ and $|\Delta\ell| < \delta_{\text{skip}}$ **then**
        Skip batch
      else
        Apply Top-k gradient sparsification
        Quantize sparse gradients
        Update $w_k^{(r)}$ using sparse gradients
      end if
    end for
    Compute sparse weight delta $\Delta_k^{(r)}$ using threshold θ_{adaptive}
    Send $\Delta_k^{(r)}$ to server
  end for
  Server performs sparse FedAvg aggregation to compute $w^{(r+1)}$
end for

IV. Experimental Setup

We evaluate FedTinyProp on image, time-series, and audio classification. All methods use identical models and optimization settings.

A. Datasets and Preprocessing

Vision: FashionMNIST, CIFAR-10/100 with inputs normalized (dataset-specific mean/std). CIFAR uses random crop with padding and horizontal flip during training while test-time uses normalization only.
HAR [15]: multivariate inertial signals (6 activities). Signals are reshaped for 1D CNNs and standardized per feature (zero mean, unit variance). Training augments include light noise, scaling, and feature masking.
Speech Commands [16]: 1 s utterances transformed to 40-dim MFCCs (Mel Frequency Cepstral Coefficients). Features are padded/truncated to a fixed temporal length and augmented to include additive noise, time-stretch, and random masking. Class labels use a vocabulary-to-index mapping.
We use CIFAR-10 for on-device measurements because its convolutional backbone is sufficiently demanding to expose measurable latency, memory, and thermal effects on edge-device hardware. We also use CIFAR-10 to study component ablations (Dense, Fixed Top-k, Adapt-only, Skip-only, Full FedTinyProp) and to quantify controller/top-k overhead versus compute/communication savings.

B. Model Architectures

We design task-specific architectures for each dataset based on input modality, task complexity, and suitability for sparse gradient training. The models maximize accuracy per FLOP and minimize activation memory while remaining robust under adaptive sparse backpropagation. ReLU activations, batch normalization, and local pooling ensure quantization-friendly, memory-predictable computation on edge devices. Residual connections stabilize sparse gradients. Global average pooling replaces large fully connected heads to cut parameters and uplink payloads.
FashionMNIST: A lightweight CNN with two convolutional layers (3×3 kernels, ReLU), max pooling, two fully connected layers.
CIFAR-10 and CIFAR-100: A compact ResNet-8 variant with convolutional stem, three residual blocks (channels: 32, 64, 128), and global average pooling before the final dense classifier.
HAR: A 1D CNN with three convolutional blocks (kernel size 5), each followed by batch normalization, ReLU, and max pooling. Temporal features are aggregated via global average pooling and processed by two dense layers before classification.
Speech Commands: A hybrid CNN for MFCC inputs, featuring a residual 1D convolutional encoder, temporal downsampling through adaptive pooling, and a deep 1D temporal stack, followed by layer-normalized dense layers for classification.

C. Training Configuration

We train for R=100 rounds with full participation; each client runs E=1 local epoch per round, batch size 32. Optimizer: SGD with momentum, cosine decay with warmup, gradient clipping at 1.0. Initial learning rates: CIFAR 0.01, HAR 0.005, Speech 0.001, FashionMNIST 0.0005. Label smoothing with ϵ=0.1 for image/audio; weight decay 5×10^{-4} on CIFAR (none on FashionMNIST; task-appropriate on others). Sparsity control (bounds and scaling) follows the adaptive policy described earlier with a short warmup to ramp sparsity. All results use FedTinyProp with an aggressive sparsity setup ($S_{\min}$=0.5, $S_{\max}$=0.95, ζ=0.5) to emphasize efficiency. Client heterogeneity is simulated via Dirichlet label splits over K=5 clients. For C classes, client class proportions are sampled from $\mathrm{Dir}(\alpha\mathbf{1}_C)$ with a single heterogeneous regime with Dirichlet α=0.5 and used to form disjoint index subsets.

D. Baselines

We compare FedTinyProp against two dense baselines, FedAvg and FedProx, as well as the dynamic sparse training method RigL. FedAvg is the canonical dense FL method. It performs weighted averaging of client updates without sparsity or skipping. FedProx [17] extends FedAvg with a proximal term ($\mu = 0.01$) to stabilize local training under heterogeneous data. RigL [11] is a dynamic sparse training method that maintains sparse masks throughout training, updating them on a cosine schedule every 100 steps, with target sparsity chosen to match our compute budget. We select these baselines because they cover the three main axes of comparison: the canonical dense FL algorithm (FedAvg), a widely used heterogeneity-robust variant (FedProx), and a state-of-the-art dynamic sparse training method (RigL). Other works from the related literature (e.g., DGC, FedPAQ, SpaFL, SparsyFed) also explore sparse or communication-efficient federated training. We do not include them here because our comparison focuses on a dense FL baseline (FedAvg), a heterogeneity-aware dense baseline (FedProx), and a sparse training baseline (RigL) under the same local training setup. On the server side, aggregation is identical for all methods: updates are averaged in proportion to each client's local data size n_k, without any server optimizer or momentum.

E. Evaluation Metrics

We focus on three core aspects of resource efficiency critical to TinyML systems: compute cost, memory consumption, and communication overhead, highlighting both dataset-specific behaviors and general trends. Per round, averaged across participating clients, we report test accuracy, training FLOPs accounting for sparse backward passes, peak training memory, uplink and downlink bytes (including sparse indices and quantized values), sparsity and compression ratio (dense update size divided by transmitted size), skipped batches and the resulting effective compute ratio and per-round wall-clock latency with a timing breakdown (forward, backward, Top-k, quantization, controller, optimizer step, and delta packing), including controller/Top-k/non-backprop overhead percentages. We report uplink communication volume as the total bytes sent from clients to the server per round, including indices and quantized nonzero update values, aggregated over all participating clients. Download communication is measured as a dense full-model broadcast.

F. Reproducibility and Environment

Experiments are implemented in Flower with custom modules for adaptive sparse backpropagation and local training control. Experiments are first validated on a Windows 11 workstation (Python 3.11.4, PyTorch 2.6.0, Flower 1.16.0) and then deployed without modification to a Raspberry Pi 4 Model B (8 GB RAM) for edge-device evaluation.

V. Results

A. Accuracy vs. Efficiency Trade-off

To assess the efficiency gains of FedTinyProp, we compare its final accuracy, average sparsity, and average compute cost across training rounds. Table II summarizes results on five datasets, highlighting the trade-off between model performance and computational efficiency.

TABLE II
FINAL TEST ACCURACY, MEAN SPARSITY, AND MEAN PER-ROUND COMPUTE (GFLOPS) OVER TRAINING ROUNDS FOR DENSE, FEDPROX, RIGL, AND FEDTINYPROP.

Dataset	Method	Accuracy	Sparsity	GFLOPs
FashionMNIST	Dense	84%	0%	12.8
	FedProx	84%	0%	12.8
	RigL	79%	88%	7.3
	FedTinyProp	83%	67%	2.9
CIFAR-10	Dense	79%	0%	29.2
	FedProx	79%	0%	29.2
	RigL	78%	88%	17.4
	FedTinyProp	76%	74%	6.3
CIFAR-100	Dense	57%	0%	29.2
	FedProx	56%	0%	29.2
	RigL	51%	88%	17.4
	FedTinyProp	51%	65%	7.4
HAR [15]	Dense	81%	0%	12.2
	FedProx	78%	0%	12.2
	RigL	75%	57%	6.8
	FedTinyProp	77%	80%	6.5
SpeechCommands [16]	Dense	66%	0%	15.2
	FedProx	64%	0%	15.2
	RigL	61%	91%	8.6
	FedTinyProp	61%	89%	7.9

FedTinyProp reduces average compute cost while retaining competitive accuracy. On vision benchmarks, FedTinyProp cuts mean per-round compute (in GFLOPS) by about 4–5$\times$ at moderate-to-high sparsity (65–89%). On CIFAR-10, this yields a 4.6$\times$ compute reduction with a 3-point accuracy drop. On CIFAR-100, the compute reduction remains substantial (29.2$\rightarrow$7.4 GFLOPs) with a larger accuracy gap of 6 percentage points, reflecting the harder task. For the two time-series

and audio datasets, the savings from roughly halving GFLOPS are smaller but remain consistent. For HAR, FedTinyProp maintains high accuracy; for SpeechCommands, FedTinyProp matches RigL.

B. Communication and Memory Savings

Table III summarizes communication and memory efficiency. FedTinyProp consistently reduces uplink communication, with upload savings ranging from 60.5% (CIFAR-10) to 85.8% (SpeechCommands). These reductions correlate with high update sparsity (74–89%) and strong compression ratios from 3 to 8.7×, indicating that sparse deltas and quantized values substantially shrink the transmitted payload. Memory savings are workload-dependent. On HAR and SpeechCommands, peak activation/gradient memory drops by about 45% at the best rounds, consistent with sparse backpropagation retaining only a subset of intermediate buffers. On CIFAR-10, the maximum peak-memory reduction is smaller with 2.2%, suggesting that in this configuration the dominant gains come from communication and skipped computation rather than activation storage. As shown in Table III, batch skipping improves efficiency by reducing the number of backward passes, thereby lowering compute and limiting the occurrence of large gradient buffers and delta updates. Overall, sparsity and quantization primarily reduce communication volume, while sparse backpropagation and skipping reduce peak memory and compute, yielding combined efficiency improvements without changing the server downlink broadcast.

TABLE III
COMMUNICATION AND CLIENT-MEMORY EFFICIENCY OF FEDTINYPROP OVER 100 ROUNDS. UPLOAD SAVED AND COMPRESSION REFER TO THE MEAN CLIENT→SERVER UPLINK PAYLOAD (SPARSE INDICES + QUANTIZED VALUES) RELATIVE TO DENSE. MAX MEM SAVED IS THE MAXIMUM OBSERVED REDUCTION IN PER-CLIENT PEAK ACTIVATION + GRADIENT MEMORY VS. DENSE. SKIPPED BATCHES IS THE TOTAL NUMBER OF SKIPPED MINI-BATCHES ACROSS ALL CLIENTS.

Dataset	Upload saved	Max mem saved	Skipped batches	Compression
HAR	64.9%	44.9%	10,597	4.92×
CIFAR-10	60.5%	2.2%	74,179	3.07×
SpeechCommands	85.8%	45.2%	126,611	8.72×

C. On-device Results

Table IV summarizes Raspberry Pi training runs comparing FedTinyProp to the dense baseline on CIFAR-10 at a shorter on-device operating point (R=30, E=1, batch size 8). FedTinyProp reduces compute cost by 6.6× GFLOPs, lowers process RAM and latency slightly, and keeps the device cooler, while maintaining 91% sparsity and 0.6 effective compute ratio. Accuracy is somewhat lower than the dense baseline, but efficiency gains on edge-device hardware are substantial.

D. Quantifying Overhead vs. Savings

The sparse and adaptive mechanisms of FedTinyProp introduce additional work (gradient statistics, controller decisions, and top-k selection). We decompose the local per-batch compute into (i) sparse backpropagation, (ii) top-k selection, (iii) controller updates (EMA/thresholds/decisions), and (iv) skipping, which reduces the number of backward passes executed. We instrument per-round wall-clock time and report controller and top-k overhead alongside savings in FLOPs relative to the dense baseline (Table V).

TABLE IV
RASPBERRY PI TRAINING BENCHMARK ON CIFAR-10.

	FedTinyProp	Dense
Final Accuracy	52%	60%
Avg. GFLOPs	4.4	29.2
Avg. Process RAM (RSS)	915.1	973.1
Avg. Temperature [°C]	72.7	79.3
Avg. Latency [ms]	41.5	44.0

TABLE V
OVERHEAD VS. SAVINGS ON CIFAR-10 (FINAL ROUND). OVERHEAD IS REPORTED AS A FRACTION OF TOTAL PER-ROUND WALL-CLOCK TIME; SAVINGS ARE RELATIVE TO DENSE. THE FINAL COLUMN REPORTS SAVINGS-TO-OVERHEAD RATIO (SAVED FLOPS % DIVIDED BY NON-BACKPROP OVERHEAD %).

Method	Ctrl.	Top-k	Non-bp	Saved FLOPs / Non-bp
Fixed Top-k	0.0%	18.4%	25.2%	1.6
Adapt-only	0.0%	18.6%	25.5%	2.8
Skip-only	0.1%	0.0%	8.4%	5.8
Full FedTinyProp	0.1%	10.7%	18.3%	4.6

Table V makes the overhead–savings trade-off explicit. The controller is effectively negligible because it performs only lightweight per-layer bookkeeping (EMA updates, thresholding, and sparsity/skip decisions). The dominant overhead is Top-k selection: about 18.6% for Fixed Top-k and Adapt-only, and 10.7% for Full FedTinyProp. Despite this additional work, the savings-to-overhead ratios remain favorable. Full FedTinyProp removes about 85% of dense FLOPs (29.2→4.4 GFLOPs) while incurring 18.3% non-backprop overhead, yielding 4.6× saved-FLOPs-per-overhead. Adapt-only shows a smaller but still favorable ratio (2.8×), reflecting that adaptive sparsity alone eliminates substantial backprop work but still pays Top-k selection costs. Skip-only achieves the largest ratio (5.8×) because it avoids Top-k entirely. However, it does not reduce uplink or memory because updates remain dense.

We analyze CIFAR-10 component ablations (Fixed Top-k, Skip-only, Full FedTinyProp) to isolate the effect of adaptive sparsity, Top-k selection, and skipping on compute savings and wall-clock behavior. Figure 2 further decomposes the per-round wall-clock time into its dominant contributors. The Pareto breakdown shows that round time is dominated by the training kernels (forward and backward passes), which account for the majority of per-round latency across variants. The primary non-backprop overhead comes from Top-k selection: it is substantial for Top-k-based variants and largely absent for Skip-only, explaining why wall-clock improvements do not always scale proportionally with FLOP reductions. Quantization adds a smaller but noticeable cost, while gradient-norm computation and controller updates remain negligible.

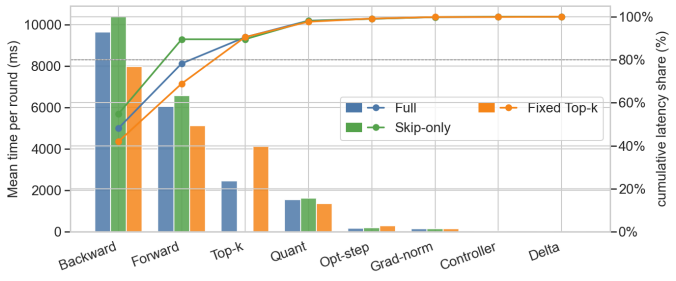

Fig. 2. CIFAR-10 component ablations: overhead and latency contributor Pareto plot.

Overall, Fig 2 indicates that FedTinyProp's adaptivity logic is lightweight, and that the main latency trade-off is between reduced backprop compute and the additional cost of sparse update selection, consistent with the overhead trends reported in Table V.

VI. Discussion

Across five benchmarks, the results show a consistent compute–accuracy trade-off: FedTinyProp reduces client-side training compute by about 2–5× while maintaining accuracy within 1–6 points of dense baselines. Table II shows that the largest compute reductions were achieved for the vision benchmarks (FashionMNIST, CIFAR-10). More moderate compute reductions were found for smaller time-series and audio workloads (HAR, SpeechCommands), reflecting differences in model size and the amount of redundant gradient computation that can be pruned. On Raspberry Pi for CIFAR-10, FedTinyProp reduces client compute 6.6× and slightly lowers process RAM, average latency, and temperature, with an 8-point accuracy drop (Table IV).

The latency analysis clarifies why wall-clock improvements may not match FLOP reductions. Tab V and Fig 2 show that the controller cost is negligible, while Top-k selection constitutes the dominant non-backprop overhead. Consequently, optimizing sparse selection (or amortizing it with kernel-level primitives) is a key pathway to converting FLOP savings into proportional runtime gains.

We identify two practical limitations. First, although uploads shrink substantially, the server still broadcasts a dense full model each round, so downlink can dominate total bandwidth for larger models. Second, Top-k selection remains the main runtime overhead and can limit wall-clock speedups. Our on-device case study uses Raspberry Pi. Porting to microcontroller-class targets, which typically provide only sub-MB memory and much tighter runtime constraints, is left for future work.

VII. Conclusion

FedTinyProp combines batch-wise adaptive sparse backpropagation, optional batch skipping, and sparse delta aggregation to reduce client compute, memory, and uplink cost in FL. Across FashionMNIST, CIFAR-10/100, HAR, and SpeechCommands, we observed substantial client-side training efficiency gains by about 2–5× while accuracy maintained within 1–6 points of baselines. Future work may focus on reducing Top-k overhead, addressing downlink communication bandwidth, and validating microcontroller-class targets.

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Aguera y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2017, pp. 1273–1282.

[2] P. Kairouz, H. B. McMahan *et al.*, "Advances and open problems in federated learning," *Foundations and Trends in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[3] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[4] W.-H. Lim *et al.*, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[5] F. Sattler, K.-R. Wiedemann, K.-R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-iid data," *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[6] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[7] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2020, pp. 2021–2031.

[8] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[9] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Sparse binary compression: Towards distributed deep learning with minimal communication," in *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[10] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 3299–3308.

[11] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in *Proceedings of the 37th International Conference on Machine Learning (ICML)*. PMLR, 2020, pp. 2943–2952.

[12] M. Rüb, A. Sikora, and D. Mueller-Gritschneder, "Advancing on-device neural network training with tinypropv2: Dynamic, sparse, and efficient backpropagation," in *Proceedings of the 2024 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2024. [Online]. Available: https://doi.org/10.1109/IJCNN60899.2024.10650122

[13] M. Kim, W. Saad, M. Debbah, and C. S. Hong, "Spafl: Communication-efficient federated learning with sparse models and low computational overhead," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. [Online]. Available: https://arxiv.org/abs/2406.00431

[14] A. Guastella, L. Sani, A. Iacob, A. Mora, P. Bellavista, and N. D. Lane, "Sparsyfed: Sparse adaptive federated training," in *International Conference on Learning Representations (ICLR)*, 2025. [Online]. Available: https://arxiv.org/abs/2504.05153

[15] J. L. Reyes-Ortiz, D. Anguita, A. Ghio, L. Oneto, and X. Parra, "Human activity recognition using smartphones," UCI Machine Learning Repository, 2013, dOI: https://doi.org/10.24432/C54S4K.

[16] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *ArXiv e-prints*, Apr. 2018. [Online]. Available: https://arxiv.org/abs/1804.03209

[17] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," in *Proceedings of the 3rd MLSys Conference*. MLSys, 2020, pp. 429–450.