FEDTINY: PRUNED FEDERATED LEARNING TOWARDS SPECIALIZED TINY MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural network pruning has been a well-established compression technique to enable deep learning models on resource-constrained devices. The pruned model is usually specialized to meet specific hardware platforms and training tasks (defined as deployment scenarios). However, existing pruning approaches rely heavily on training data to trade off model size, efficiency, and accuracy, which becomes ineffective for federated learning (FL) over distributed and confidential datasets. Moreover, the memory- and compute-intensive pruning process of most existing approaches cannot be handled by most FL devices with resource limitations.

In this paper, we develop FedTiny, a novel distributed pruning framework for FL, to obtain specialized tiny models for memory- and computing-constrained participating devices with confidential local data. To alleviate biased pruning due to unseen heterogeneous data over devices, FedTiny introduces an adaptive batch normalization (BN) selection module to adaptively obtain an initially pruned model to fit deployment scenarios. Besides, to further improve the initial pruning, FedTiny develops a lightweight progressive pruning module for local finer pruning under tight memory and computational budgets, where the pruning policy for each layer is gradually determined rather than evaluating the overall deep model structure. Extensive experimental results demonstrate the effectiveness of FedTiny, which outperforms state-of-the-art baseline approaches, especially when compressing deep models to extremely sparse tiny models.

1 INTRODUCTION

Deep neural networks (DNNs) have achieved great success in the past decade. However, the huge computational cost and storage overhead limit the usage of DNNs on resource-constrained devices. Neural network pruning has been a well-known solution to improve hardware efficiency (Janowsky, 1989; Han et al., 2015). The core of neural network pruning is to remove insignificant parameters from a DNN and determine specialized subnetworks for different hardware platforms and training tasks (defined as deployment scenarios). To achieve better accuracy, most pruning approaches rely heavily on training data to trade off model size, efficiency, and accuracy (Han et al., 2015; Louizos et al., 2018; Yu et al., 2018; Molchanov et al., 2019b; Singh & Alistarh, 2020), which, unfortunately, becomes ineffective when dealing with confidential training datasets distributed over resource-constrained devices.

Recent success in federated learning enables collaborative training across distributed devices with confidential local datasets (Li et al., 2020b). Instead of uploading local data, federated learning aggregates on-device knowledge by iteratively updating local model parameters at the server. While successful, federated learning cannot determine the specialized pruned model for participating devices without training data. To address this issue, (Xu et al., 2021) proposed to decouple the pruning process under federated environments, where a large-size model is first pruned on the server and then fine-tuned on devices. However, since most pruning algorithms require the guide from the data distribution, without access to device-side training data, the server-side pruning leads to significant bias in the pruned subnetwork, especially under heterogeneous (non-iid) local data distributions. To mitigate such bias issues, some recent research pushes pruning operations to devices (Shao et al., 2019; Li et al., 2021; Munir et al., 2021; Liu et al., 2021; Jiang et al., 2022). As shown in Figure 1 left, either a full-size model or a coarse-pruned model will be finer-pruned based on the updated

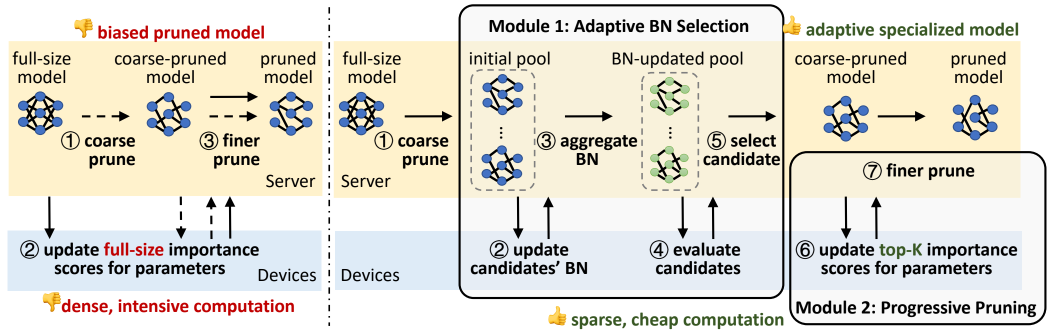

Figure 1: Overview of FedTiny for the specialized tiny model in federated learning. *Left*: Existing federated pruning approaches push pruning operations to devices. Either a full-size model (solid arrow) or a coarse-pruned model (dash arrow) is finer-pruned under dense and intensive local computation, suffering biased pruning. *Right*: FedTiny introduces two key modules, the adaptive batch normalization module and the progressive pruning module, to adaptively search coarse- and finer-pruned specialized models to fit deployment scenarios with sparse and cheap local computation.

importance scores from devices. The importance scores for all parameters need to store in memory, which is infeasible for resource-constrained devices with limited memory budgets. Moreover, without any interaction with the device side, the initial model through server-side coarse pruning still suffers from the bias issue, requiring extra efforts in later finer pruning to find the optimal subnetwork. Such negative impact becomes more challenging when pruning towards an extremely tiny subnetwork, as the biased initial subnetwork can deviate significantly from the optimal structure, resulting in poor accuracy (Evci et al., 2020).

To address the above challenges, in this paper, we develop a novel distributed pruning framework for federated learning, named FedTiny. Depending on the deployment scenarios, *i.e.*, participating hardware platforms and training tasks, FedTiny can obtain specialized tiny models using distributed and confidential datasets on participating devices. Besides, FedTiny allows devices with tight memory and computational budgets to participate in the resource-intensive pruning process by reconfiguring interactions between the server and devices. As shown in Figure 1 right, FedTiny introduces two key modules: the adaptive batch normalization (BN) selection module and the progressive pruning module. To avoid the negative impact of biased initial pruning, we introduce the *adaptive BN selection module* to identify a specialized coarse-pruned model by indirectly pruning at devices, where devices only evaluate the server-side pruning. It should be mentioned that evaluating a pruned model is much cheaper than training and pruning. The local evaluation is feedback to the server through batch normalization (BN) parameters. Since BN can effectively measure local data distribution with very few parameters (Ioffe & Szegedy, 2015), this module guides the initial pruning with little computation and communication cost. Besides, contrary to prior research using important scores of all parameters in a full-size model for finer pruning, the *progressive pruning module* is developed to iteratively adjust the model structure with sparse and cheap local computation. Inspired by RigL (Evci et al., 2020), devices only rate partial model parameters (e.g., a single layer) at a time, where the top-K importance scores are stored locally and uploaded to the server, significantly reducing memory, computation, and communication costs.

To demonstrate the effectiveness of FedTiny, we evaluate FedTiny on ResNet18 (He et al., 2016) and VGG11 (Simonyan & Zisserman, 2014) with four image classification datasets (CIFAR-10, CIFAR-100, CINIC-10, and SVHN). Extensive experimental results suggest that FedTiny achieves much higher accuracy with a lower level of memory and computational costs than state-of-the-art baseline approaches. Especially in a low-density regime (Hoefler et al., 2021) from 10^{-2} to 10^{-3}, FedTiny gets a small loss of accuracy, while other baselines suffer from sharp drops in accuracy. Moreover, FedTiny achieves top-one accuracy of 85.23% with the $0.014\times$ FLOPs and $0.03\times$ memory footprints of ResNet18 (He et al., 2016), which outperforms the best baseline, which gets 82.62% accuracy with $0.034\times$ FLOPs and $0.51\times$ memory footprints.

2 Related Work

Neural Network Pruning. Neural network pruning has been a well-known technique to remove redundant parameters of a DNN for model compression, which can trace back to the late 1980s (Mozer & Smolensky, 1988; LeCun et al., 1989; Janowsky, 1989). Most existing pruning approaches focus on the trade-off between accuracy and sparsity in the *inference* stage. A typical pruning process first calculates the importance scores of all parameters in a well-trained DNN and then removes parameters with lower scores. The importance scores can be derived based on the weight magnitudes (Janowsky, 1989; Han et al., 2015), the first-order Taylor expansion of the loss function (Mozer & Smolensky, 1988; Molchanov et al., 2019a), the second-order Taylor expansion of the loss function (LeCun et al., 1989; Hassibi & Stork, 1992; Molchanov et al., 2019b), and other variants (Louizos et al., 2018; Yu et al., 2018; Singh & Alistarh, 2020).

Another line of recent research on neural network pruning focuses on improving the efficiency of the *training* stage, which can be divided into two categories. One is pruning at initialization, *i.e.*, pruning the original full-size model before training. The pruning policy can be determined by evaluating the connection sensitivity (Lee et al., 2018), Hessian-gradient product (Wang et al., 2019), and synaptic flow (Tanaka et al., 2020) of the original model. Since such pruning does not involve the training data, the pruned model is not specialized for the training task, resulting in biased performance. The other category is dynamic sparse training (Mocanu et al., 2018; Dettmers & Zettlemoyer, 2019; Evci et al., 2020). The pruned model structure is iteratively adjusted throughout the training process while maintaining the pruned model size at the desired sparsity. However, the pruning process is to adjust the model structure in a large search space, requiring memory-intensive operations, which is infeasible for resource-constrained devices. Although RigL Evci et al. (2020) tries to reduce memory consumption, it needs to compute gradients for all parameters, which is computationally expensive and may lead to straggling issues in federated learning

Federated Neural Network Pruning. Federated learning has attracted great attention by enabling collaborative training across distributed and confidential datasets (Li et al., 2020b). Classical federated learning, represented by FedAvg (McMahan et al., 2017), collects the locally updated on-device models rather than the raw data at the server for private knowledge sharing. Since data is locally stored and cannot be shared, the aforementioned pruning approaches that rely on training data cannot be used in federated learning. Enlighten by pruning at initialization, (Xu et al., 2021) prunes the original full-size model at the server, and fine-tunes at devices with their local data. Existing pruning at initialization approaches, such as SNIP (Lee et al., 2018), GraSP (Wang et al., 2019) and SynFlow (Tanaka et al., 2020), can be directly converted to server-side pruning. However, server-side pruning usually results in significantly biased pruned models, especially for heterogeneous (non-iid) local data distributions.

To mitigate such bias, recent research pushes pruning operations under federated settings to devices. By locally training a full-size model, SCBF (Shao et al., 2019) dynamically discards the unimportant channels on devices. Such local training with a full-size model is assigned to a part of devices in FedPrune to guide pruning based on the updated activations (Munir et al., 2021). Besides, LotteryFL (Li et al., 2021) iteratively prunes a full-size model on devices with a fixed pruning rate to find a personalized local subnetwork. However, the above research suffers from large memory and computational costs on the device side, because devices need to locally compute the importance scores of all parameters. Although PruneFL (Jiang et al., 2022) reduces the local computational cost by finer pruning a coarse-pruned model rather than a full-size model, it still requires a large local memory footprint to record the updated important scores of all parameters in the full-size model. The coarse-pruned model still suffers from bias issues in the server-side pruning. Therefore, existing federated neural network pruning fails to obtain a specialized tiny model without bias and memory-/compute-budget concerns, and we develop FedTiny to achieve this.

3 Proposed FedTiny

This section introduces the proposed FedTiny. We first describe the problem statement, followed by our design principles. Accordingly, we present two key modules in FedTiny: the adaptive BN selection module and the progressive pruning module.

3.1 Problem Statement

We consider a typical federated learning setting, where K devices collaboratively train a neural network with their corresponding local datasets $\mathcal{D}_k, k \in \{1, 2, \ldots, K\}$. All devices have limited memory and computing resources. Given a large neural network with dense parameters $\mathbf{\Theta}$, we aim to find a specialized subnetwork with sparse parameters $\boldsymbol{\theta}$ and mask $\boldsymbol{m}$ on dense parameters to achieve the optimal prediction performance for federated learning. The sparse parameters are derived by applying a mask to the dense parameters: $\boldsymbol{\theta} = \mathbf{\Theta} \odot \boldsymbol{m}$ ($\boldsymbol{m} \in \{0,1\}^{|\mathbf{\Theta}|}$). During the training, the density d of the sparse mask $\boldsymbol{m}$ cannot exceed the target density d_{target}, which is determined by the limitations of devices' memory resources. We formulate the problem as a constrained optimization problem:

$$\begin{aligned} \min_{\boldsymbol{\theta}, m} \quad & \sum_{k=1}^{K} L(\boldsymbol{\theta}, \boldsymbol{m}, \mathcal{D}_k), \\ \text{s.t.} \quad & d \leq d_{target} \end{aligned} \tag{1}$$

where $L(\boldsymbol{\theta}, \boldsymbol{m}, \mathcal{D}_k)$ denotes the loss function for local dataset $\mathcal{D}_k$ on the k-th device.

3.2 Design Principles

As shown in Figure 1 left, existing federated neural network pruning faces two main challenges, bias in coarse pruning and intensive memory consumption in finer pruning. To address these challenges, we propose a FedTiny. The overview of FedTiny is illustrated in Figure 1 right, which consists of two key modules: the adaptive BN selection module and the progressive pruning module.

The adaptive BN selection module (Steps 2-5 in Figure 1 right) aims to derive an adaptive coarse-pruned structure on the server and alleviate bias in the coarse pruning due to unseen heterogeneous data over devices. In this module, the devices first collaboratively update the BN measurements for all candidate models from coarse pruning, and then the server selects one less biased candidate model as the initial coarse-pruned model based on device evaluations.

The progressive pruning module (Steps 6-7 in Figure 1 right) further improves the coarse-pruned model by finer pruning at resource-constrained devices, significantly reducing the on-device memory footprint and computational cost. In this module, the devices only maintain the top-K importance scores of the pruned parameters. Based on the average importance scores, the server grows and prunes parameters to produce a new model structure. After iterative growing and pruning, the model structure progressively approaches the optimal structure.

In the following, we provide detailed descriptions of the adaptive BN selection module and the progressive pruning module, respectively.

3.3 Adaptive Batch Normalization Selection

It is critical to address the bias issue in the coarse-pruned model, as the highly biased pruned structure requires more resources and time to adjust to the optimal structure, especially in the low-density regime. One possible approach is to send a set of pruned structure candidates to the devices and let devices select the least biased model from the candidate pool. We call this approach vanilla selection (He et al., 2018; Liu et al., 2020). However, recent research (Li et al., 2020a) shows that pruned model performance varies before and after fine-tuning, which makes the pruned structure candidate selected before fine-tuning not necessarily the best one after fine-tuning. Such an issue could be exaggerated in the federated settings as the heterogeneous data distribution over devices may further increase the discrepancy of pruned model performance in fine-tuning.

To address this issue, we introduce adaptive BN selection in FedTiny. Adaptive BN selection updates the BN measurements for candidate models before evaluation, aiming to derive a less biased coarse-pruned structure. Batch Normalization (BN) (Ioffe & Szegedy, 2015) provides measurements for data distribution across devices, which provides a representation of on-device data and thus guides the pruning processing. Moreover, BN transformation is calculated as part of the forward pass at the device, which does not incur too much memory and computational cost. The BN transformation is upon the following transformation on i-th input x_i in each batch,

$$\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{2}$$

where ϵ is a small constant. μ and σ are calculated based on the mean and standard deviation of the batch in training and kept fixed in testing.

In this module, BN measurements are updated in the forward pass on devices before evaluation to select a less biased-coarse pruned candidate. Specifically, after coarse pruning on full-size parameters $\boldsymbol{\Theta}$ with different strategies, the server obtains an initial pool consisting of C candidate models with their sparse parameters $\boldsymbol{\theta}^{(c)}$ and the corresponding masks $\boldsymbol{m}^{(c)}$, where $\boldsymbol{\theta}^{(c)} = \boldsymbol{\Theta} \odot \boldsymbol{m}^{(c)}$, for $c \in \{1, \ldots, C\}$. For each candidate model, we set different pruning ratios for each layer while keeping overall density $d \leq d_{target}$. Appendix H shows the details of candidate pool generation. Devices first fetch all candidate models. Note that the communication cost is low due to the ultra-low network density, which will be discussed in Appendix C. Then, each device (say the k-th) samples a development dataset from local data, $\hat{\mathcal{D}}_k \subset \mathcal{D}_k$, freezes all parameters and updates the means $\mu_k^{(c)}$ and standard deviations $\sigma_k^{(c)}$ of BN layers in the c-th candidate model. Next, the server aggregates all local BN measurements from devices to obtain new global BN measurements for each candidate model, *i.e.*, $\mu^{(c)} = \sum_{k=1}^{K} \frac{|\hat{\mathcal{D}}_k|}{\sum_{k=1}^{K} |\hat{\mathcal{D}}_k|} \mu^{(c)}$ and $\sigma^{(c)} = \sum_{k=1}^{K} \frac{|\hat{\mathcal{D}}_k|}{\sum_{k=1}^{K} |\hat{\mathcal{D}}_k|} \sigma^{(c)}$, for $c \in \{1, \ldots, C\}$. After that, each device updates global BN measurements $\mu^{(c)}, \sigma^{(c)}$ for c-th candidate model. Considering Equation 1, we let devices calculate evaluation loss for each BN-updated candidate model with their on-device data, and let the server select the candidate model with the lowest average loss as the coarse-pruned model. Since there are no gradients and the backward pass in adaptive BN selection, the additionally introduced computation cost is acceptable. The algorithm of the adaptive BN selection module is illustrated in Appendix B.1.

3.4 Progressive Pruning

Given a coarse-pruned model from the above module, we introduce progressive pruning to further fine-prune the model for better performance. We propose the progressive pruning module with two improvements: 1) except top-K importance scores, the most importance scores can be discarded to save memory space; 2) partial model parameters (*e.g.*, a single layer) is adjusted per rounds instead of the entire model to avoid intensive computation. FedTiny utilizes a growing-pruning adjustment on the model structure while maintaining the sparsity. Specifically, the server grows the pruned parameters and prunes the same number of unpruned parameters to adjust the model structure. Denote a_t^l as the number of parameters that will be grown and pruned on layer l at the t-th iteration. To guide the growing and pruning on the server, each device only trains the sparse model and computes the Top-a_t^l gradients for pruned parameters, which keeps the low memory footprint and computational cost in the resource-constrained device. Furthermore, to reduce intensive computation, FedTiny divides the model structure into several blocks and prunes a block in one round.

In detail, each device (say the k-th) first downloads global sparse model parameter $\boldsymbol{\theta}_t$ with mask $\boldsymbol{m}_t$ as their local parameters $\boldsymbol{\theta}_t^k$ in the t-th iteration, and applies SGD with sparse gradients:

$$\boldsymbol{\theta}_{t+1}^k = \boldsymbol{\theta}_t^k - \eta_t \nabla L(\boldsymbol{\theta}_t^k, \boldsymbol{m}_t, \mathcal{B}_t^k) \odot \boldsymbol{m}_t, \tag{3}$$

where η_t is the learning rate, $\mathcal{B}_t^k$ is a batch of sample from the local dataset $\mathcal{D}_k$, and $\nabla L \odot \boldsymbol{m}_t$ denotes the sparse gradients for the sparse parameter $\boldsymbol{\theta}_t^k$. After E iterations of local SGD, each device calculates the top-a_t^l gradients for pruned parameters on each layer l with a batch of samples. In detail, clients create a buffer in the memory to store a_t^l gradients. When a gradient is calculated and the buffer is full, if its magnitude is larger than the smallest magnitude in the buffer, this gradient will be pushed into a buffer and the gradient with the smallest magnitude will be discarded. Otherwise, this gradient will be discarded. In this manner, clients only need $O(a_t^l)$ memory space to store the gradients. We denote $\tilde{\boldsymbol{g}}_t^{k,l}$ as the top-a_t^l gradients of pruned parameter with the largest magnitude on k-th device:

$$\tilde{\boldsymbol{g}}_t^{k,l} = \mathrm{TopK}\left(\boldsymbol{g}_t^{k,l}, a_t^l\right), \tag{4}$$

where $\mathrm{TopK}(\boldsymbol{v}, k)$ is threshold function, the elements of $\boldsymbol{v}$ whose absolute value is less than the k-th largest absolute value are replaced with 0, and $\boldsymbol{g}_t^{k,l}$ is the gradients of pruned parameters on layer

l. Next, the server aggregates sparse parameters and gradients to get average parameter and average gradients $\tilde{\boldsymbol{g}}_t^l = \sum_{k=1}^{K} \frac{|\mathcal{D}_k|}{\sum_{k=1}^{K} |\mathcal{D}_k|} \tilde{\boldsymbol{g}}_t^{k,l}$. Then, the server grows a_t^l pruned parameters with the largest averaged gradients magnitude on each layer l. After that, the server prunes a_t^l unpruned parameters (excluding the parameters just grown) with the smallest magnitude on each layer l. According to growing and pruning, the server generates a global model with a new model structure and FedTiny starts fine-tuning the new global model. After ΔR rounds of fine-tuning, FedTiny will prune the model again unless the round number reaches the rounds at which to stop pruning R_{stop}. The algorithm of the progressive pruning module is illustrated in Appendix B.2.

4 Experiments

In this section, we conduct comprehensive experiments on FedTiny. First, we introduce the experiment setting and compare FedTiny with other baselines. Second, we conduct the ablation study to demonstrate the effeteness of the adaptive BN selection module and the progressive pruning module. Finally, we investigate the impact of the candidate pool size and pruning schedule in FedTiny.

4.1 Experimental Setup

We evaluate FedTiny on image classification tasks with four datasets, CIFAR-10, CIFAR-100 (Krizhevsky et al., 2009), CINIC-10 (Darlow et al., 2018), and SVHN (Netzer et al., 2011) datasets on ResNet18 (He et al., 2016) and VGG11 (Simonyan & Zisserman, 2014) models. We consider $K = 10$ devices in total. For all datasets, we generate various non-IID partitions on devices from Dirichlet distribution with $\alpha = 0.5$, following Luo et al. (2021). We train the models for 300 FL rounds on the CIFAR-10, CIFAR-100, and CINIC-10 datasets and 200 rounds on the SVHN dataset. Each round includes 5 local epochs. The mini-batch size is set as 64. We also evaluate FedTiny in the scenarios with a large number of clients and EfficientNet Tan & Le (2019), which will be discussed in Appendix G and Appendix I.

FedML (He et al., 2020) framework is used to implement our FedTiny. In the adaptive BN selection module, we first set the size of the candidate pool to 50 and then change the candidate pool size in Section 4.4. We first set the ratio of the development dataset as 0.1. In the progressive pruning module, we divide ResNet18 and VGG11 into five blocks and prune one block in each round. The order in which the server selects a block is backward, i.e., from the output layer to the input layer. We also evaluate pruning a single layer and pruning the entire model per round in section 4.4. The pruning number is set as $a_t^l = 0.15(1 + cos\frac{t\pi}{R_{stop}E})n^l$ for layer l that will be pruned at the t-th iteration, where n^l is the number of unpruned parameters in l-th layer. For layer l that will not be pruned in the t-th iteration, $a_t^l = 0$. We do not prune the batch normalization layer, bias, input layer, and output layer because they affect model output directly.

We involve the following baseline approaches in the study. Since SNIP (Lee et al., 2018) and PruneFL (Jiang et al., 2022) require some data for coarse pruning, we assume that the server provides a public one-shot dataset $\mathcal{D}_s$ for pretraining. All baselines start with a model pre-trained with the one-shot dataset $\mathcal{D}_s$ on the server.

- **SNIP** (Lee et al., 2018) prunes model based on connection sensitivity at initialization with the one-shot dataset $\mathcal{D}_s$ on the server.
- **SynFlow** (Tanaka et al., 2020) prunes model by iteratively conserving synaptic flow on the server before training.
- **FL-PQSU** (Xu et al., 2021) prunes model in a one-shot manner based on l_1-norm on the server before training. FL-PQSU also includes quantization and selective update parts, but we only use the pruning part in FL-PQSU.
- **PruneFL** (Jiang et al., 2022) uses a powerful and trusted device to initially prune the model and applies finer pruning (adaptive pruning) on the sparse model based on full-size averaged gradients. But all devices are resource-constrained in our setting. Therefore, we let PruneFL get the initial pruned model on the server with the public one-shot dataset $\mathcal{D}_s$.
- **LotteryFL** (Li et al., 2021) iteratively prunes dense model with a fixed pruning rate on devices and re-initializes the pruned model with the initial values.

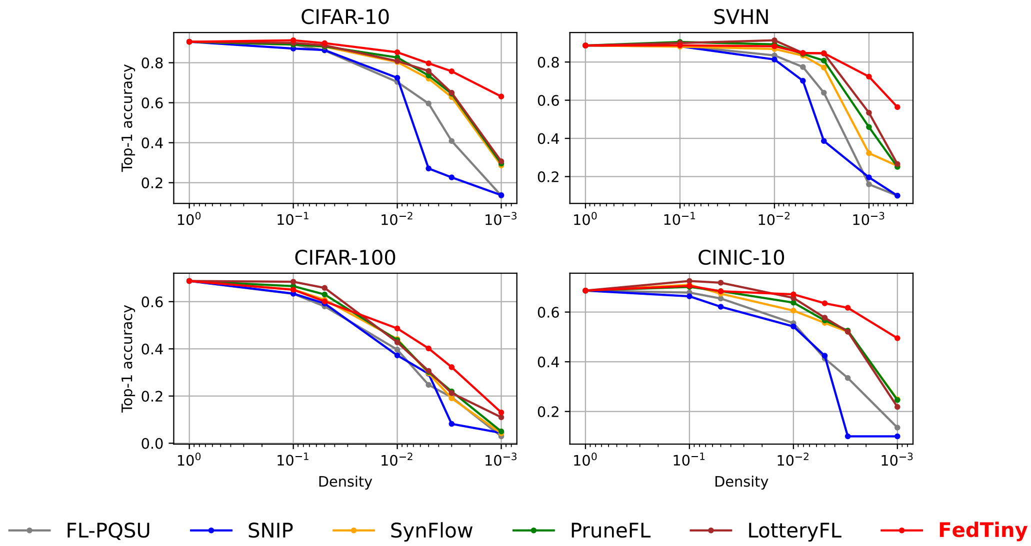

Figure 2: Top-1 accuracy of different pruning approaches in federated learning. We compare the proposed FedTiny with baselines on the four datasets with different densities. FedTiny outperforms the baselines, especially in the extremely low-density regimes ($< 10^{-2}$).

We include SNIP, SynFlow, and FL-PQSU to confirm that pruning at initialization is not the optimal design choice when the local data are invisible.[1] For SNIP, we apply iterative pruning instead of one-shot pruning with $\mathcal{D}_s$ as (Tanaka et al., 2020) shown. Similarly, we let SynFlow prune the model at initialization to the target density in an iterative manner. For SNIP and SynFlow, we set 100 pruning epochs on the server at initialization, refer to (Tanaka et al., 2020). For FL-PQSU, which is originally structured pruning, we change it to unstructured pruning since all the other baselines are unstructured pruning frameworks. LotteryFL (Li et al., 2021) is designed for personalized federated learning, so the model structures are different among devices. Since we attempt to find an optimal structure for all devices as in Equation 1, we let LotteryFL iteratively prune the global model instead of on-device models to ensure the same model structure for each device. Since LotteryFL, PruneFL, and our FedTiny are iteratively pruning during training, we use the same pruning schedule for these frameworks, where the framework does $\Delta R = 10$ rounds of fine-tuning between two finer pruning. And framework stops pruning and continues fine-tuning after $R_{stop} = 100$ rounds. For PruneFL, we set the pruning number a_t^l to be the same as in FedTiny. All baselines will apply uniform sparsity distribution for layer-wise pruning rate setting. We exclude the FL pruning approaches that are infeasible for memory-constrained FL. For example, FedPrune (Munir et al., 2021) and SCBF (Shao et al., 2019) require powerful devices to continuously process the dense models.

4.2 FedTiny vs. Baselines

In order to show the performance of FedTiny under different densities, we compare baselines and FedTiny on four datasets (CIFAR-10, CIFAR-100, CINIC-10, and SVHN) with ResNet18. As shown in Figure 2, FedTiny outperforms the other baselines in the low-density regime ($d_{target} < 10^{-2}$). which demonstrates less bias in FedTiny, as indicated by the adaptive BN selection module. Besides, FedTiny is also competitive with a high density ($d_{target} > 10^{-2}$). Although LotteryFL can partially outperform FedTiny under high density, it requires more resources to process dense models on devices. SNIP performs badly in low density because SNIP tends to remove nearly all parameters in some layers. Moreover, the pruned model in SNIP highly depends on the samples on the server, which increases bias due to non-IID.

[1]We implement these frameworks based on their open-source implementations `https://github.com/ganguli-lab/Synaptic-Flow`.

Density	Method	ResNet18 Top-1 Accuracy	ResNet18 Training FLOPs Peak	ResNet18 Memory Footprints	VGG11 Top-1 Accuracy	VGG11 Training FLOPs Peak	VGG11 Memory Footprints
1	FedAvg	**0.9048**	**1x(8.33E13)**	**90.91MB**	**0.8696**	**1x(4.09E13)**	**1033.33MB**
0.01	FL-PQSU	0.7038	0.014x	2.75MB	0.475	0.017x	20.96MB
	SNIP	0.7245	0.014x	2.76MB	0.3481	0.017x	20.98MB
	SynFlow	0.8034	0.014x	2.75MB	0.5803	0.017x	20.92MB
	PruneFL	**0.8262**	0.34x	46.58MB	**0.6204**	0.34x	526.87MB
	LotteryFL	0.8083	1x	90.91MB	0.6183	1x	1033.33MB
	FedTiny	**0.8523**	0.014x	2.79MB	**0.7883**	0.017x	20.95MB
0.005	FL-PQSU	0.5961	0.008x	2.01MB	0.232	0.012x	11.99MB
	SNIP	0.2711	0.009x	1.98MB	0.2409	0.012x	12.00MB
	SynFlow	0.7206	0.008x	2.00MB	0.4376	0.012x	11.95MB
	PruneFL	0.736	0.34x	46.19MB	**0.4956**	0.34x	522.31MB
	LotteryFL	**0.7586**	1x	90.91MB	0.4376	1x	1033.33MB
	FedTiny	**0.7972**	0.009x	2.03MB	**0.7534**	0.012x	11.98MB
0.001	FL-PQSU	0.1352	0.004x	1.22MB	0.1	0.008x	4.71MB
	SNIP	0.1377	0.004x	1.19MB	0.1	0.008x	4.72MB
	SynFlow	0.2862	0.004x	1.19MB	0.2531	0.008x	4.72MB
	PruneFL	0.2955	0.336x	45.72MB	**0.2692**	0.339x	518.71MB
	LotteryFL	**0.307**	1x	90.91MB	0.2634	1x	1033.33MB
	FedTiny	**0.6311**	0.004x	1.17MB	**0.5944**	0.008x	4.71MB

Table 1: Top-1 accuracy and training cost on ResNet18 and VGG11 with different densities on the CIFAR-10 dataset. We report Training FLOPs Peak, the maximum training FLOPs in a single round, and the memory footprints in devices. The best performance of Top-1 accuracy with the same density is represented in red, and the second best metric is marked in blue. Performances of FedAvg are in bold for reference. All cost measurements are for one device in one pruning round.

To show the efficiency of FedTiny, we measure the cost of training ResNet18 and VGG11 with various densities on the CIFAR-10 dataset. The number of floating point operations (FLOPs) is used to measure the computational cost for each device. We use training FLOPs peak per round to evaluate whether devices suffer from intensive computation in a single round. Memory Footprints are related to memory cost in deployment. As shown in Table 1, the proposed FedTiny outperforms other baselines in accuracy while getting the lowest levels of FLOPs and memory footprint. Appendix J discusses how training FLOPs and memory footprints are calculated.

4.3 Ablation Study

This section discusses the effectiveness of each module in FedTiny via ablation studies. We evaluate vanilla selection, adaptive BN selection, progressive pruning after vanilla selection, and FedTiny on the CIFAR-10 dataset with the VGG11 model. Figure 3 shows the results of each module working individually. We have the following three findings. First, both the adaptive BN selection module and progressive pruning module improve the performance in vanilla selection, indicating the effectiveness of these two modules. Second, a coarse-pruned model from adaptive BN selection faces a drop in accuracy compared to FedTiny, indicating that there are still some biases in the selected coarse-pruned model, and the progressive pruning module can remove them. Last, the progressive pruning module with vanilla selection reaches the same level of accuracy compared to FedTiny with the high density ($< 10^{-2}$). However, it suffers from severe degradation of accuracy in the low-density regime ($> 10^{-2}$), which suggests that the progressive pruning module only removes the bias to a certain extent and it must be combined with the adaptive BN selection module in the low-density regime. Therefore, independently using the adaptive BN selection module and progressive pruning module can improve performance, but the improvement is limited. The combination of the two modules, *i.e.*, FedTiny, achieves the best prediction performance with the tiny model.

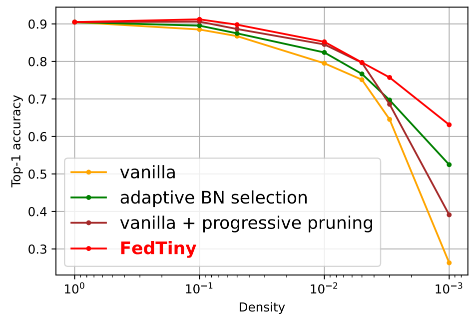

Figure 3: Ablation studies the two key modules in FedTiny: the adaptive BN selection module and the progressive pruning module. We compare vanilla selection, adaptive BN selection, progressive pruning with vanilla selection, and FedTiny. We test the ResNet18 model on the CIFAR-10 dataset with various densities.

4.4 Impact of Candidate Pool Size and Pruning Schedule

Although a larger candidate pool provides more choices for selection, it brings more communication costs in the adaptive BN selection module. So, we want to find an optimal pool size that can trade off the accuracy and communication costs. Therefore, We evaluate FedTiny on VGG11 with different pool sizes to find an optimal pool size . The result shows that the optimal pool size is $C^* = \frac{0.1}{d_{target}}$ for specific density d_{target}, where the communication cost in adaptive BN selection module is as low as 20% to a full-size VGG11 model and FedTiny can receive a relatively good accuracy. A larger pool size $> C^*$ slightly improves accuracy but incurs much higher communication costs. Moreover, with the optimal pool size C^*, the extra FLOPs in adaptive BN selection are less than one round of sparse training. Because there are hundreds of rounds in training, the extra computation cost is neglectable. Figure 4 in the Appendix shows the top-1 accuracy and communication cost in the adaptive BN selection module for various pool sizes under different densities on VGG11 with the CIFAR-10 dataset. Table C in the Appendix shows the FLOPs in adaptive BN selection with different target densities.

Although layer-wise adjustment in progressive pruning reduces the computation cost in one round, it may slow down the convergence speed. To determine the best pruning granularity and pruning frequency, we evaluate FedTiny on VGG11 with different pruning granularities (one layer per round, one block per round, and the entire model per round) and different pruning frequencies. If the pruning granularity is too small (e.g., layer-wise pruning), the model structure will converge slowly, and the optimal structure cannot be achieved with limited training resources. But high updating granularity leads to more intensive computation in one round. We find that pruning a block per round is an optimal choice for the progressive pruning module. Moreover, sequentially choosing blocks to prune in backward order (from the output layer to the input layer) gets better results than forwarding order since the gradient propagation is backward, and we use gradients to adjust the model structure. Table 3 in the Appendix shows the top-1 accuracy of various pruning schedules under different densities on VGG11 with the CIFAR-10 dataset.

5 Conclusion

This paper develops a novel distributed pruning framework, called FedTiny. FedTiny enables memory-efficient local training and determines specialized tiny models in federated learning for different deployment scenarios (participating hardware platforms and training tasks). FedTiny addresses the bias issues and intensive computation and memory issues that existing federated pruning research suffers. FedTiny introduces two key modules: the adaptive batch normalization (BN) selection module and the progressive pruning module, for adaptive coarse-pruning and lightweight finer-pruning, respectively. Extensive experimental results demonstrated that FedTiny outperforms state-of-the-art baseline approaches, especially when compressing full-size deep models to the low-density regime.

6 Ethics Statement

We develop a distributed pruning framework to enable federated learning on different hardware platforms and training tasks and to produce specialized tiny models. The framework mitigates the privacy threats of directly learning from distributed confidential data across devices. This work has the potential to facilitate a wide range of on-device intelligent applications without privacy concerns.

7 Reproducibility Statement

To facilitate reproducibility of experiment results, we provide the experimental setups in Section 4.1, Appendix ,B.2 and Appendix J, including the datasets, hyper-parameters, implementation details, and evaluation measurements. Furthermore, we point out that the settings and hyperparameters when the settings differ from the default settings. Those are sufficient for reproducibility. Moreover, we plan to open-source the codes in the future.

References

Luke N Darlow, Elliot J Crowley, Antreas Antoniou, and Amos J Storkey. Cinic-10 is not imagenet or cifar-10. *arXiv preprint arXiv:1810.03505*, 2018.

Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pp. 2943–2952. PMLR, 2020.

Masafumi Hagiwara. Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, volume 1, pp. 351–354. IEEE, 1993.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5, 1992.

Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 784–800, 2018.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989.

Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. Model pruning enables efficient federated learning on edge devices. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. Snip: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2018.

Ang Li, Jingwei Sun, Binghui Wang, Lin Duan, Sicheng Li, Yiran Chen, and Hai Li. Lotteryfl: Empower edge intelligence with personalized and communication-efficient federated learning. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 68–79. IEEE, 2021.

Bailin Li, Bowen Wu, Jiang Su, and Guangrun Wang. Eagleeye: Fast sub-net evaluation for efficient neural network pruning. In *European conference on computer vision*, pp. 639–654. Springer, 2020a.

Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020b.

Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4876–4883, 2020.

Shengli Liu, Guanding Yu, Rui Yin, and Jiantao Yuan. Adaptive network pruning for wireless federated learning. *IEEE Wireless Communications Letters*, 10(7):1572–1576, 2021.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

Mi Luo, Fei Chen, Dapeng Hu, Yifan Zhang, Jian Liang, and Jiashi Feng. No fear of heterogeneity: Classifier calibration for federated learning with non-iid data. *Advances in Neural Information Processing Systems*, 34:5972–5984, 2021.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pp. 1273–1282. PMLR, 2017.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

P Molchanov, S Tyree, T Karras, T Aila, and J Kautz. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017-Conference Track Proceedings*, 2019a.

Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11264–11272, 2019b.

Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. *Advances in neural information processing systems*, 1, 1988.

Muhammad Tahir Munir, Muhammad Mustansar Saeed, Mahad Ali, Zafar Ayyub Qazi, and Ihsan Ayyub Qazi. Fedprune: Towards inclusive federated learning. *arXiv preprint arXiv:2110.14205*, 2021.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

Rulin Shao, Hui Liu, and Dianbo Liu. Privacy preserving stochastic channel-based federated learning with neural network pruning. *arXiv preprint arXiv:1910.02115*, 2019.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression. *Advances in Neural Information Processing Systems*, 33:18098–18109, 2020.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105–6114. PMLR, 2019.

Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2019.

Wenyuan Xu, Weiwei Fang, Yi Ding, Meixia Zou, and Naixue Xiong. Accelerating federated learning for iot in big data analytics with pruning, quantization and selective updating. *IEEE Access*, 9:38457–38466, 2021.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 9194–9203, 2018.

Appendix

A Effect of Progressive Pruning

We apply a growing-pruning adjustment to the model structure while maintaining the sparsity. Specifically, the server grows the pruned parameters and prunes the same number of unpruned parameters to adjust the model structure. Like prior works (Janowsky, 1989; Han et al., 2015), we prune the parameters with the smallest magnitude. On the one hand, the squared value of weights can be viewed as weight power, and magnitude pruning removes the parameters with the least power to improve the computational efficiency Hagiwara (1993). On the other hand, magnitude pruning can remove trivial parameters and increase the generalization ability of the network.

After pruning, we grow the pruned parameters with the largest gradient magnitude, like RigL (Evci et al., 2020). If the gradient magnitude of a pruned parameter is large, this parameter is important and should not be pruned. Growing pruned parameters with high gradients will reduce the loss quickly, which can help the model structure to reach optimal faster in Equation 1.

B Algorithms

B.1 Adaptive BN Selection

Algorithm 1 Adaptive BN selection

Input: C coarse-pruned candidate models with sparse parameters $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(C)}$ and their corresponding masks $\boldsymbol{m}^{(1)}, \dots, \boldsymbol{m}^{(C)}$ on server, K clients with local development dataset $\hat{\mathcal{D}}_1, \dots, \hat{\mathcal{D}}_K$.
Output: the less biased coarse-pruned model with parameters $\boldsymbol{\theta}_0$ and its corresponding mask $\boldsymbol{m}_0$.

// Client-side
for $k = 1$ to K **do**
  fetch sparse parameters $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(C)}$ and their corresponding masks $\boldsymbol{m}^{(1)}, \dots, \boldsymbol{m}^{(C)}$.
  for $c = 1$ to C **do**
    $\mu_k^{(c)}, \sigma_k^{(c)} \leftarrow$ forward pass on $\boldsymbol{\theta}^{(c)}$ with dataset $\hat{\mathcal{D}}_k$ to calculate local BN measurements.
  end for
  upload $\mu_k^{(1)}, \dots, \mu_k^{(C)}$ and $\sigma_k^{(1)}, \dots, \sigma_k^{(C)}$.
end for
// Server-side
for $c = 1$ to C, **Server do**
  $\mu^{(c)} \leftarrow \sum_{k=1}^{K} \frac{|\hat{\mathcal{D}}_k|}{\sum_{k=1}^{K} |\hat{\mathcal{D}}_k|} \mu_k^{(c)}$.
  $\sigma^{(c)} \leftarrow \sum_{k=1}^{K} \frac{|\hat{\mathcal{D}}_k|}{\sum_{k=1}^{K} |\hat{\mathcal{D}}_k|} \sigma_k^{(c)}$.
end for
// Client-size
for $k = 1$ to K **do**
  fetch global BN measurements $\mu^{(1)}, \dots, \mu^{(C)}$ and $\sigma^{(1)}, \dots, \sigma^{(C)}$.
  for $c = 1$ to C **do**
    $\mu_k^{(c)}, \sigma_k^{(c)} \leftarrow \mu^{(c)}, \sigma^{(c)}$. // each candidate model configures global BN measurements.
    $s_k^{(c)} \leftarrow L(\boldsymbol{\theta}^{(c)}; \boldsymbol{m}^{(c)}; \hat{\mathcal{D}}_k)$. // calculate the loss as evaluation metrics.
  end for
  upload $s_k^{(1)}, \dots, s_k^{(C)}$.
end for
// Server-side
for $c = 1$ to C **do**
  $s^{(c)} \leftarrow \sum_{k=1}^{K} \frac{|\hat{\mathcal{D}}_k|}{\sum_{k=1}^{K} |\hat{\mathcal{D}}_k|} s_k^{(c)}$.
end for
$c^* \leftarrow \mathrm{argmin}_c(s^{(c)})$ // select the candidate model with the lowest loss.
return $\boldsymbol{\theta}_0, \boldsymbol{m}_0 \leftarrow \boldsymbol{\theta}^{(c^*)}, \boldsymbol{m}^{(c^*)}$

B.2 Progressive Pruning

Algorithm 2 Progressive pruning

Input: initial coarse-pruned parameters $\boldsymbol{\theta}_0$ with mask $\boldsymbol{m}_0$, K clients with local dataset $\mathcal{D}_1, \ldots \mathcal{D}_K$, iteration number t, learning rate η_t, pruning number a_t^l for each layer l, the number of local iterations per round E, the number of rounds between two pruning operation ΔR, and the rounds at which to stop pruning R_{max},.
Output: a well-trained model with sparse θ_t and adjusted mask m_t

$t \leftarrow 0$.
while no stop criteria **do**
  // Client-side
  for $k = 1$ to K **do**
    fetch sparse parameters $\boldsymbol{\theta}_t$ and it corresponding mask $\boldsymbol{m}_t$.
    for $i = 0$ to $E - 1$ **do**
      $\boldsymbol{\theta}_{t+i+1}^k \leftarrow \boldsymbol{\theta}_{t+i}^k - \eta_{t+i} \nabla L(\boldsymbol{\theta}_{t+i}, \boldsymbol{m}_t, \mathcal{B}_{t+i}^k) \odot \boldsymbol{m}_t$. // the mask keeps the same in each round
    end for
    upload $\boldsymbol{\theta}_{t+E}$
    if $t \mod \Delta RE = 0$ and $t \leq ER_{max}$ **then**
      for each layer l in model **do**
        $\tilde{\boldsymbol{g}}_t^{k,l} \leftarrow$ compute top-a_t^l sparse gradients for pruned parameters as Equation 4.
      end for
      upload all $\tilde{\boldsymbol{g}}_t^{k,l}$ for each layer l.
    end if
  end for
  // Server-side
  $\boldsymbol{\theta}_{t+E} \leftarrow \sum_{k=1}^{K} \frac{|\mathcal{D}_k|}{\sum_{k=1}^{K} |\mathcal{D}_k|} \boldsymbol{\theta}_{t+E}^k$.
  if $t \mod \Delta RE = 0$ and $t \leq ER_{max}$ **then**
    for each layer l in model **do**
      $\tilde{\boldsymbol{g}}_t^l \leftarrow \sum_{k=1}^{K} \frac{|\mathcal{D}_k|}{\sum_{k=1}^{K} |\mathcal{D}_k|} \tilde{\boldsymbol{g}}_t^{k,l}$
      $\boldsymbol{I}_{grow}^l \leftarrow$ record the a_t^l pruned indices with the largest absolute value in $\tilde{\boldsymbol{g}}_t^l$.
      $\boldsymbol{I}_{drop}^l \leftarrow$ record the a_t^l unpruned indices with smallest weight magnitude in $\boldsymbol{\theta}_{t+E}$.
      $\boldsymbol{m}_{t+E}^l \leftarrow$ adjust $\boldsymbol{m}_t^l$ by negating the masks whose indices in $\boldsymbol{I}_{grow}^l$ and $\boldsymbol{I}_{drop}^l$.
    end for
    $\boldsymbol{\theta}_{t+E} \leftarrow \boldsymbol{\theta}_{t+E} \odot \boldsymbol{m}_{t+E}$
  else
    $\boldsymbol{m}_{t+E} \leftarrow \boldsymbol{m}_t$.
  end if
  $t \leftarrow t + E$
end while

C The Impact of Candidate Pool Size

We do the experiments on CIFAR-10 datasets with VGG11 model with different pool size and densities. The experiments are shown in Figure 4, an interesting observation is that the pool size $C^* = \frac{0.1}{d_{target}}$ is the optimal value for a given target density d_{target}, since larger pool sizes do not improve much accuracy. And with optimal pool size C^*, the communication costs in the adaptive BN selection module are about 20% of a full-size VGG11. As federated learning needs to transfer the model for a large number of rounds. A one-time 20% model-size communication is very small compared with the entire communication cost.

We also calculate the extra FLOPs for the adaptive BN selection module with optimal pool size, as shown in Table C. the extra FLOPs in adaptive BN selection are less than one round of sparse training. Since federated learning usually involves more than one hundred rounds of training, the

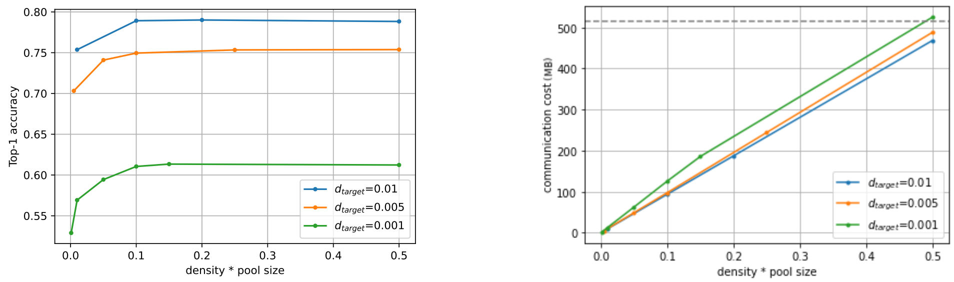

Figure 4: The performance and cost for sparse VGG11 model with different densities and pool sizes. *Left*: The effect of pool size on top-1 accuracy under different densities. *Right*: The effect of pool size on communication costs in the adaptive BN selection module under different densities. The gray dash line is the size for a full-size VGG11 model.

Density	Pool Size	Extra FLOPs in selection	Training FLOPs in one round
0.01	10	9.15E+10	6.86E+11
0.005	20	1.3E+11	4.92E+11
0.001	100	3.42E+11	3.56E+11

Table 2: Extra FLOPs in the adaptive BN selection model

extra computational overhead is neglectable. Therefore, we argue that the overhead introduced by adaptive BN selection is marginal.

D THE IMPACT OF PRUNING SCHEDULE

The pruning schedule determines the pruning granularity and pruning frequency in the progressive pruning module. For pruning granularity, we choose to prune one layer per round (Layer), pruning one block per round (Block), and prune the entire model per round (Entire). We divide the model into five blocks to prune, as shown in Figure 5. Moreover, we control the pruning frequencies by setting different interval rounds ΔR between two pruning. Since the pruning times of each layer should be the same, we set corresponding stopping rounds R_{stop} for different pruning granularity and ΔR. Moreover, We also wonder if the order in which layers or blocks are selected to prune has any effect.

Therefore, we evaluate sparse VGG11 with different pruning granularities, pruning frequencies, and selection orders. Table 3 shows the results. If the pruning granularity is too small, the model structure will converge slowly, and the optimal structure cannot be achieved with limited training resources. Thus, in our experiment, we find the optimal choice is pruning one block per round, which provides a stable and fast model structure convergence. Another interesting observation is that lower pruning frequency may get higher accuracy since it has more stable model parameters before pruning. Moreover, small pruning granularity leads to high pruning frequency with limited rounds; high pruning granularity leads to more intensive computation in one round. Therefore, pruning a block per round may be an optimal choice for the progressive pruning module. Another interesting observation is that selecting a layer or block to prune in backward order is better than in forwarding order. The reason is that gradient propagation is backward, and we use gradients to adjust the model structure.

E THE IMPACT OF DATA HETEROGENEITY

Neural network pruning requires training data to determine the proper model structure. Due to resource-constrained devices, the server cannot push the dense model to devices. Therefore, the server needs to coarsely prune to produce the initial pruned model. Due to privacy concerns in federated learning, the server cannot know the data distributions for all clients. So, in the existing

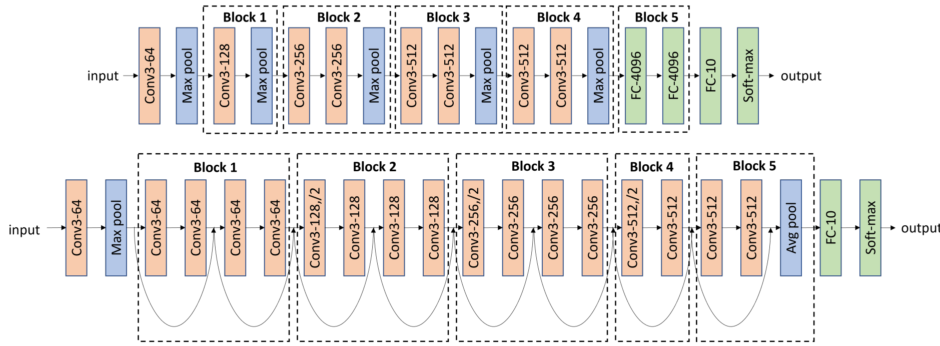

Figure 5: Partition of blocks of the VGG11 (top) and ResNet18 (bottom) models.

Granularity	$\Delta R/R_{stop}$	Density 0.01	Density 0.005	Density 0.001
Layer	5/100	0.7623	0.7034	0.447
Layer (b)	5/100	0.7894	0.7343	0.5871
Block	10/100	0.7697	0.7179	0.5721
Block (b)	10/100	0.7883	0.7534	0.6311
Block (b)	5/50	0.7675	0.7263	0.6113
Entire	50/100	0.772	0.7395	0.6244
Entire	25/50	0.7583	0.7043	0.5944

Table 3: Accuracy for FedTiny with different adjustment schedules when progressive pruning VGG11 with the CIFAR-10 dataset. The default selection order is forwarding order, i.e., from the input layer to the output layer, and b donates backward order.

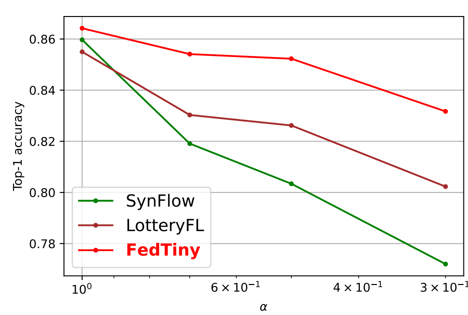

Figure 6: Top-1 accuracy of different pruning approaches on various non-iid degrees. Lower α indicates a higher Non-IID degree.

methods, the server only coarsely prunes the model based on the pretrain dataset or the data from some trusted clients. It makes the dataset used for pruning different from the dataset used for fine-tuning, which causes bias in coarse pruning. Therefore, our strategy is to use adaptive BN selection to select one pruned model with less bias.

To show the impact of the data heterogeneity, we set different non-iid degrees by using different α in the Dirichlet distribution. Lower α indicates a higher Non-IID degree. We do experiments on the CIFAR-10 dataset with ResNet18 with 1% density. The experiments are shown in Figure 6. Our experiments show that 1) the performance of the existing pruning methods (e.g., SynFlow, LotteryFL) in Federated Learning will be significantly degraded given a higher non-iid degree; 2) Our proposed FedTiny mitigates the bias in pruning and achieves the best performance compared with the existing pruning methods.

Dataset	SynFlow	LotteryFL	Small Model	FedTiny
CIFAR-10	0.8034	0.8262	0.8019	0.8523
CINIC-10	0.6057	0.6379	0.5578	0.6712
SVHN	0.8683	0.8927	0.8395	0.8826
CIFAR-100	0.4413	0.4373	0.4277	0.4865

Table 4: The Top-1 accuracy for ResNet18 with 1% density and a small model on a different dataset.

Density	SynFlow	LotteryFL	small models	FedTiny
0.01	0.8034	0.8262	0.8019	0.8523
0.005	0.7206	0.736	0.7201	0.7972
0.003	0.6279	0.6453	0.6921	0.7572
0.001	0.2862	0.2955	0.6158	0.6311

Table 5: The Top-1 accuracy for ResNet18 with various densities and small models on CIFAR-10 dataset.

F THE PERFORMANCE OF SMALL MODEL

Although FedTiny can outperform other baselines in the very sparse model, like 1% density, the accuracy suffers from a drop compared to the full-size models. The small model can be considered as a baseline in this case. Therefore, we design the experiments on small models. We train a small network with 3 convolutional layers. First, we evaluate the small network with a similar number of parameters to ResNet18 with 1% density on different datasets. Second, we evaluate the small network with a similar number of parameters to ResNet18 with different densities on CIFAR-10. We also choose SynFlow and LotteryFL as references. The experiment result is shown in Table 4 and Table 5. The experimental results show that the small network is competitive compared to other baselines. However, our proposed FedTiny achieves much better performance compared with the small network, which demonstrates the advantage of FedTiny.

G THE IMPACT OF LARGE CLIENT NUMBER

To evaluate the performance of federated pruning in scenarios with a large number of clients. We conduct federated learning with 100 clients to verify the scalability of our proposed FedTiny. Only 10 clients are selected in each round. On the CIFAR-10 dataset with ResNet18 in 1% density, after 500 rounds of training, FedTiny gets 71.12% top-1 accuracy. For other baselines, SynFlow gets 59.25% accuracy, PruneFL gets 62.08% accuracy and LotteryFL gets 56.80% accuracy. FedTiny still achieves the best performance compared with the existing pruning methods.

H CANDIDATE POOL GENERATION

Given target density d_{target}, server outputs candidates in the form of layer-wise pruning rate vectors $(d^1, d^2, \ldots, d^L)$ for L-layer model based on Uniform Noise (UN) strategies. We derive the density d^l for the l-th layer by adding the target density target with random noise e^l, i.e., $d^l = d_{target} + e^l$. A candidate can be added to the candidate pool only if its total density d satisfies $d \leq d_{target}$. After that, server can get a candidate pool $\{\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(C)}\}$ with mask $\{m^{(1)}, m^{(2)}, \ldots, m^{(C)}\}$

I THE PERFORMANCE OF EFFICIENTNET

Since we focus on specialized tiny models, we evaluate the proposed FedTiny on EfficientNetTan & Le (2019), which is a state-of-the-art neural network model for small devices. We conduct the experiments on CINIC-10 and set the density as 1%. As shown in the table 6, FedTiny largely outperforms the existing methods (SynFlow, PruneFL, LotteryFL), which is similar to the result in our paper. The experimental results suggest our proposed FedTiny can be generalized to different models and outperforms the existing methods.

Full-size	SynFlow	PruneFL	LotteryFL	FedTiny
0.725	0.6546	0.6809	0.6661	0.7096

Table 6: Top-1 accuracy for EfficientNet with 1% density on CINIC-10 dataset.

J Calculating FLOPs and Memory for Models

J.1 Compression Schemes

The storage for a matrix contains two parts, value, and position. And compression goal is to reduce the storage of the positions of non-zero values in the matrix. Assuming we want to store the positions of m non-zeros value with b bit-width in a sparse matrix M. Matrix M has n elements and $n_r \times n_c$ shape. For different densities $d = m/n$, we apply different schemes to represent matrix M. We use o bits to represent the position of m non-zeros value and denote the overall storage as s.

- For density $d \in [0.9, 1]$, **dense** scheme is applied, i.e. $s = n * b$.
- For density $d \in [0.3, 0.9)$, **bitmap** (BM) is applied, which stores a map with n bits, i.e. $o = n, s = o + mb$.
- For density $d \in [0.1, 0.3)$, we apply **coordinate offset** (COO), which stores elements with its absolute offset and it requires $o = m\lceil \log_2 n \rceil$ extra bits to store position. Therefore, the overall storage is $s = o + mb$
- For density $d \in [0., 0.1)$, we apply **compressed sparse row** (CSR) and **compressed sparse column** (CSC) depending on size. It uses column and row index to store the position of elements and $o = m\lceil \log_2 n_c \rceil + n_r\lceil \log_2 m \rceil$ bits are needed for CSR. The overall storage is $s = o + mb$

For tenor, we only compress the two dimensions with the highest length. With the above strategy, we can further calculate the storage and memory of the parameters in the network.

J.2 Storage of Model

For each tensor or matrix parameter in the model, we identify its density and use the corresponding compression scheme J.1 to represent it. The storage for hyper-parameters is omitted since it is negligible.

J.3 The Memory Footprint of Training Models

We estimate training memory footprint as the combination of parameters, activations, gradients of activations, and gradients of parameters. The memory of parameters is equal to the storage of parameters. And we estimate the memory of activation by taking the maximum value of multiple measurements. For simplicity, we set the memory of gradients of activations to be equal to the memory of activations. We omit the memory of hyper-parameters and momentum. Assuming the memory for dense and sparse parameters are M_d^p and M_s^p respectively, and the memory for activations is M^a, the overall training memory for each algorithm would be the following:

- **FedAvg and LotteryFL** These methods need to train a dense model, so the memory for gradients of parameters is approximate to M_d^p. The training memory footprint is about $2M_d^p + 2M^a$.
- **FL-PQSU, SNIP, and SynFlow** These methods train a sparse static model, so the memory for gradients of parameters is approximate to M_s^p. The training memory footprint is about $2M_s^p + 2M^a$.
- **PruneFL** It requires clients to maintain dense gradients for the full-size parameters, so the memory for gradients of parameters is approximate to M_d^p. the memory footprint is about $M_d^p + M_s^p + 2M^a$.
- **FedTiny.** Since we divide the model into 5 blocks and adjust one block in one round. Moreover, we only update top-K gradients in memory to adjust model structure, so the extra

memory is used to store top-a_l^t gradients and their indices in one block. So the memory for gradients of parameters is approximate to $M_s^p + 3b\sum_l a_l^t$, where b is the bit-width. So the overall memory footprint is $2M_s^p + 2M^a + 3b\sum_l a_l^t$. Since the $3b\sum_l a_l^t$ is too small, the overall training memory footprints in FedTiny are approximate to FL-PQSU, SNIP, and SynFlow.

J.4 FLOPs of Training Models

The training FLOPs include forward pass FLOPs and backward pass FLOPs. We count the total number of operations layer by layer. In the forward pass, the layer activations are computed sequentially using the previous activations and the layer's parameters. And in the backward pass, each layer computes the activation gradient and the gradient of parameters. For simplicity, we default to **twice** as many FLOPs in the backward pass as in the forward pass. We omit the FLOPs in batch normalization and loss calculation. Assuming the local iteration number is E and the FLOPs for one forward pass with a dense and sparse model are F_d and F_s respectively, the training FLOPs peak in a round on one client is computed as follows:

- **FedAvg and LotteryFL**: These methods need to train the dense model at the beginning. Thus the peak training FLOPs peak occurs in the first round, which is $3F_dE$.
- **FL-PQSU, SNIP, and SynFlow**: These server-side pruning methods only train static sparse models during the training. Thus, peak training FLOPs are $3F_sE$.
- **PruneFL.** The training FLOPs peak occurs in the adaptive pruning (finer pruning) round, where clients maintain dense gradients for full-size parameters. Thus the backward pass is dense, and the peak FLOPs is $(2F_s + F_d)E$.
- **FedTiny.** The training FLOPs peak occurs in the finer pruning round. Since We divide the model into 5 blocks and prune one block in one round, in the finer pruning round, each client first applies the E epoch of local SGD. Then, the client sample one batch of data to calculate gradients for pruned parameters on the selected block. We find the maximum of extra FLOPs for the selected block is about $0.4F_d$. Therefore, the peak FLOPs of FedTiny is $3F_sE + 0.4F_d$. The ratio of FLOPs peak in FedTiny and FedAvg is $\frac{Fs}{F_d} + \frac{0.4}{3E}$, where the first term is equal to server-side pruning methods. The second term is as small as 0.0004 with our experiment settings. Therefore, the training FLOPs peak in our FedTiny is approximate to the server-side pruning method.