FedMef: Towards Memory-efficient Federated Dynamic Pruning

Supplementary Material

6. Proof of Theorem 1

In this section, we first introduce the internal covariate shift in CNN without batch normalization layers and then provide the proof of Theorem 1.

6.1. Internal Covariate Shift

Given a CNN model with ReLU-Conv ordering, in the l-th convolution layer, the sparse filters are represented as $\theta^l \in \mathbb{R}^{ks \times ks \times c_{in} \times c_{out}}$, where ks denotes the kernel size;c_{in} and c_{out} denote the number of input and output channels, respectively. For an input value $a^{l-1} \in \mathbb{R}^{h_{in} \times w_{in} \times c_{in}}$, the convolution operation in the l-th layer that yields the output value $a^l \in \mathbb{R}^{h_{out} \times w_{out} \times c_{out}}$ is:

$$a^l = \mathrm{Conv}(\theta^l, f(a^{l-1})), \tag{9}$$

where $f(\cdot)$ is any activation function such as ReLU, leaky ReLU, etc. It should be noted that a^{l-1} is not just an input; it is also the output of the $l-1$-th layer.

As illustrated in Figure 7, the above convolution operation can be converted into a linear multiplicity version as :

$$\psi^l = W^l x^l / c_{in}, \tag{10}$$

where weight matrix $W^l \in \mathbb{R}^{c_{out} \times (ks \cdot ks \cdot c_{in})}$ is the flattening version of the convolution filters θ^l. The i-th row of the linear weight W^l is the flattening result of the i-th filter of the original filters, θ_i^l. The linear input $x^l \in \mathbb{R}^{(ks \cdot ks \cdot c_{in}) \times (h_{out} \cdot w_{out})}$ is the stacked convolution patch from activation $f(a^{l-1})$. The resultant multiplication, ψ^l, corresponds to a reshaped version of the original output a^l. The i-th row of the linear result ψ^l is the flattening result of the i-th channel of the original output, a_i^l.

Denote the mean and variance values of the i -th filter of the original filters as $\mathbb{E}(\theta_i^l) = \mu_\theta$ and $\mathrm{Var}(\theta_i^l) = \sigma_\theta^2$. Assuming the mean and variance values of the linear input x^l are $\mathbb{E}(x^l) = \mu_x$ and $\mathrm{Var}(x^l) = \sigma_x^2$, the mean and variance of the i-th channel of output a_i^l will be :

$$\mathbb{E}(a_i^l) = \mathbb{E}(\psi_i^l) = \mathbb{E}(W_i^l)\mathbb{E}(x^l)/c_{in} = \mu_i^\theta \mu_x / c_{in}, \tag{11}$$

$$\begin{aligned} \mathrm{Var}(a_i^l) = \mathrm{Var}(\psi_i^l) &= \mathrm{Var}(W_i^l x^l)/c_{in}^2 \\ &= (\sigma_\theta^2\sigma_x^2 + \sigma_\theta^2\mu_x^2 + \mu_\theta^2\sigma_x^2)/c_{in}^2, \end{aligned} \tag{12}$$

Consider $f(\cdot)$ to be the activation function of ReLU, which implies that the input value μ_x has a positive mean. During training, the mean value of each filter θ_i^l is difficult to keep at zero. Therefore, without a batch normalization layer, the mean output from the convolution layer will not reach around zero.

6.2. Proof

Theorem *Given a CNN model structured in a ReLU-Conv sequence, and allowing the l-th convolution layer to perform operations as depicted by the forward pass in Equation 6 and NSConv in Equation 7. For the i-th channel of the activation value, $f(a_i^{l-1})$, with its mean and variance denoted as μ_f, σ_f^2. The mean and variance for the i-th channel of the output value, a_i^l, will be:*

$$\mathbb{E}[a_i^l] = 0, \quad \mathrm{Var}[a_i^l] = \gamma^2(\sigma_f^2 + \mu_f^2). \tag{13}$$

Proof. As illustrated in Figure 7, convolution operation can be converted to a linear multiplicity version as:

$$\psi^l = \hat{W}^l x^l / c_{in}, \tag{14}$$

where the weight matrix $\hat{W}^l \in \mathbb{R}^{c_{out} \times (ks \cdot ks \cdot c_{in})}$ is the flattening version of the sparse normalized convolution filters $\hat{\theta}^l$. The i-th row of the linear sparse weight $\hat{W}_i^l$ is the flattening result of the i -th filter of normalized filters, $\hat{\theta}_i^l$.

Therefore, the mean and variance of the i-th row of normalized linear weight, $\hat{W}_i^l$ are $\mathbb{E}(\hat{W}_i^l) = 0$ and $\mathrm{Var}(\hat{W}_i^l) = \gamma^2 c_{in}$. The mean and variance for the i-th of the output value will be:

$$\mathbb{E}(a_i^l) = \mathbb{E}(\psi_i^l) = \mathbb{E}(\hat{W}_i^l)\mathbb{E}(x^l)/c_{in} = 0, \tag{15}$$

$$\begin{aligned} \mathrm{Var}(a_i^l) = \mathrm{Var}(\psi_i^l) &= \mathrm{Var}(\hat{W}_i^l x^l)/c_{in}^2 \\ &= \gamma^2(\sigma_x^2 + \mu_x^2), \end{aligned} \tag{16}$$

Because the linear input x^l is the sampled version of the input activation $f(a^{l-1})$, considering randomness, the mean and variance of the linear input x^l will be $\mu_x = \mu_f$, $\sigma_x^2 = \sigma_f^2$. Therefore, we can get:

$$\mathbb{E}(a_i^l) = 0, \quad \mathrm{Var}(a_i^l) = \gamma^2(\sigma_f^2 + \mu_f^2). \tag{17}$$

6.3. Experiment Result

To assess the effectiveness of our proposed Normalized Sparse Convolution (NSConv), we conducted experiments on the CIFAR-10 dataset with the ResNet18 model in our proposed FedMef framework, with the sparsity of target parameters set to 0.9. The results of the experiment, shown

Algorithm 1 FedMef

Input: dense initialized parameters θ, K devices with local dataset $\mathcal{D}_1, \dots \mathcal{D}_K$, iteration number t, original learning rate schedule η_t, architecture adjustment schedule ξ_t^l denoting the number of adjustment parameters for each layer l, the number of local epochs per round E, the number of rounds between two adjustment ΔR, and the rounds at which to stop adjustment R_{stop}.

Output: a well-trained model with sparse parameters θ_t and specified mask m_t

$t \leftarrow 0$
$\theta_0, m_0 \leftarrow$ random prune dense initialized parameters θ
while until converge **do**
  for each device $k = 1$ to K **do**
    Fetch sparse parameters θ_t and mask m_t from the server
    for $i = 0$ to $E - 1$ **do**
      $\hat{\theta}_{t+i}^k \leftarrow$ Filter-wise Sparse Standardization as in Equation 7.
      if $t \mod \Delta RE = 0$ and $t \leq ER_{stop}$ **then**
        Calculate budget-aware learning rate β_{t+i} as in Equation 4.
        $\mu_{t+i} \leftarrow \max(\eta_{t+i}, \beta_{t+i})$
        $\theta_{t+i+1}^k \leftarrow \theta_{t+i}^k - \mu_{t+i} \nabla L_k^s(\hat{\theta}_{t+i}^k, m_t, \mathcal{D}_{t+i}^k) \odot m_t$, using scaled activation pruning
      else
        $\theta_{t+i+1}^k \leftarrow \theta_{t+i}^k - \eta_{t+i} \nabla L_k(\hat{\theta}_{t+i}^k, m_t, \mathcal{D}_{t+i}^k) \odot m_t$, using scaled activation pruning
      end if
    end for
    Upload θ_{t+E}^k to the server
    if $t \mod \Delta RE = 0$ and $t \leq ER_{stop}$ **then**
      for each layer l in model **do**
        Compute top-ξ_t^l gradients $\tilde{\boldsymbol{g}}_t^{k,l}$ for pruned parameters with a memory space of $O(\xi_t^l)$
        Upload $\tilde{\boldsymbol{g}}_t^{k,l}$ to the server
      end for
    end if
  end for

  The server does
  $\theta_{t+E} \leftarrow \sum_{k=1}^K \frac{|\mathcal{D}_k|}{\sum_{k=1}^K |\mathcal{D}_k|} \theta_{t+E}^k$
  if $t \mod \Delta RE = 0$ and $t \leq ER_{stop}$ **then**
    for each layer l in model **do**
      $\tilde{\boldsymbol{g}}_t^l \leftarrow \sum_{k=1}^K \frac{|\mathcal{D}_k|}{\sum_{k=1}^K |\mathcal{D}_k|} \tilde{g}_t^{k,l}$
      $\boldsymbol{I}_{grow}^l \leftarrow$ the ξ_t^l pruned indices with the largest absolute value in $\tilde{\boldsymbol{g}}_t^l$
      $\boldsymbol{I}_{drop}^l \leftarrow$ the ξ_t^l unpruned indices with smallest weight magnitude in θ_{t+E}
      Compute the new mask m_{t+E}^l by adjusting m_t^l based on $\boldsymbol{I}_{grow}^l$ and $\boldsymbol{I}_{drop}^l$
    end for
    $\theta_{t+E} \leftarrow \theta_{t+E} \odot m_{t+E}$ // Prune the model using the updated mask
  else
    $m_{t+E} \leftarrow m_t$
  end if
  $t \leftarrow t + E$
end while

in Figure 8, demonstrate that NSConv can achieve an effect similar to that of a Batch Normalization layer. Furthermore, the activation values of ResNet18 without normalization decrease and the distribution becomes more centralized as the layer deepens, further supporting Equations 11 and 12, which indicate that the mean and variance values will be scaled with $1/c_{in}$ and $1/c_{in}^2$, respectively.

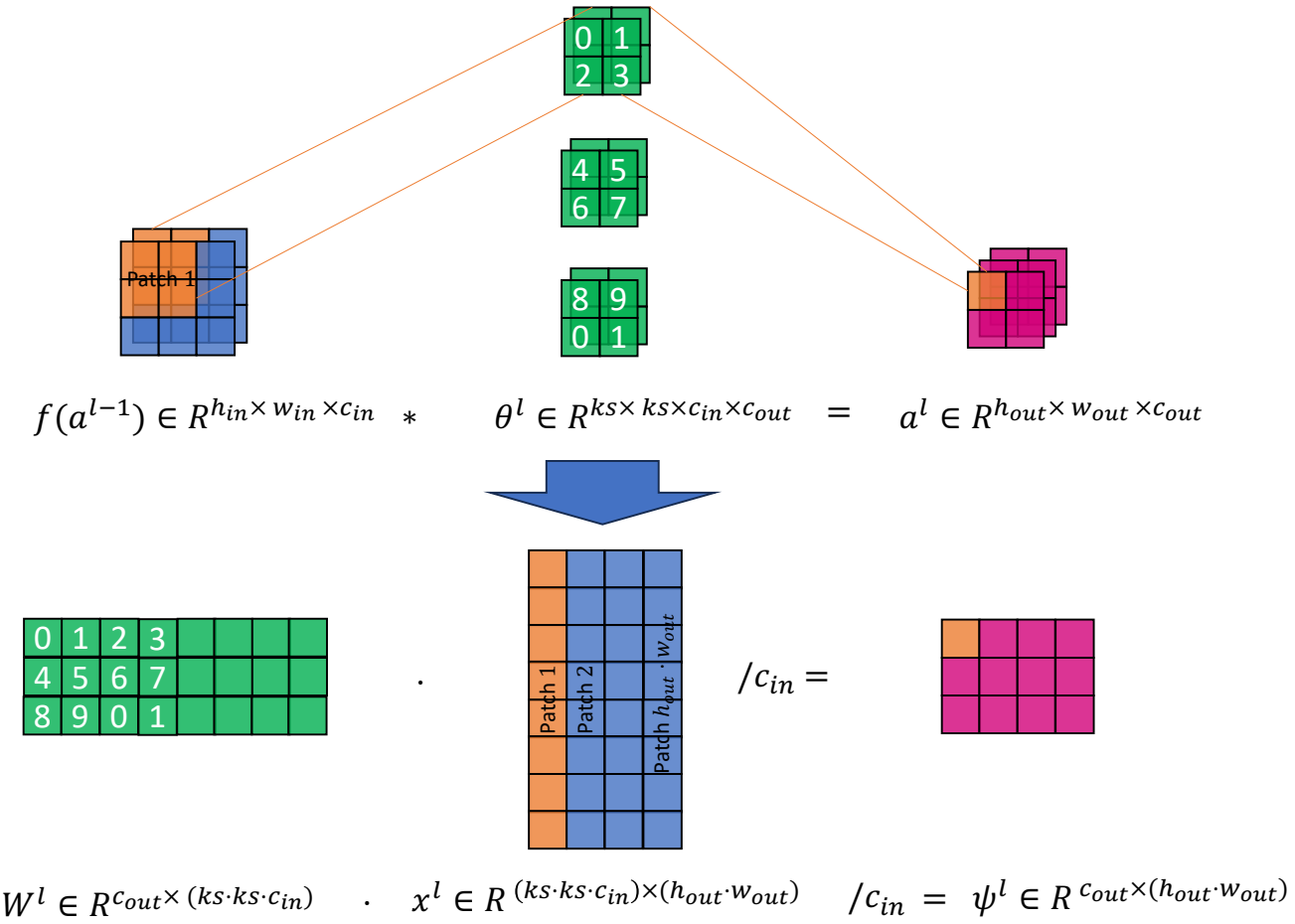

Figure 7. Illustration of transforming the convolution operation into linear multiplication: Start by flattening each filter from the convolutional filters, θ^l, and stacking them to produce the linear weight W^l. Next, stack each convolution patch from the input value $f(a^{l-1})$ to form the linear input x^l. The resultant multiplication, ψ^l, corresponds to a reshaped version of the original output a^l.

7. Memory, FLOPs and Communication Costs

In our experiments, we conducted a comparative analysis between the proposed FedMef and other baselines, focusing on training memory footprints, maximum training FLOPs per round, and communication costs per round. In this section, we first introduce the sparse compression strategies and then present the estimated calculations of the above metrics.

7.1. Compression Schemes

The storage for a matrix consists of two components, values and positions. Compression aims to reduce the storage of the positions of non-zero values in the matrix. Suppose we want to store the positions of m non-zeros value with b bit-width in a sparse matrix M. The matrix M has n elements and a shape $n_r \times n_c$. Depending on the density $d = m/n$, we apply different schemes to represent the matrix M. We use o bits to represent the positions of m nonzero values and denote the overall storage as s.

- For density $d \in [0.9, 1]$, **dense** scheme is applied, i.e. $s = n \cdot b$.
- 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 the 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 carry out reshaping before compression. This approach allows us to determine the memory needed to train the network's parameters.

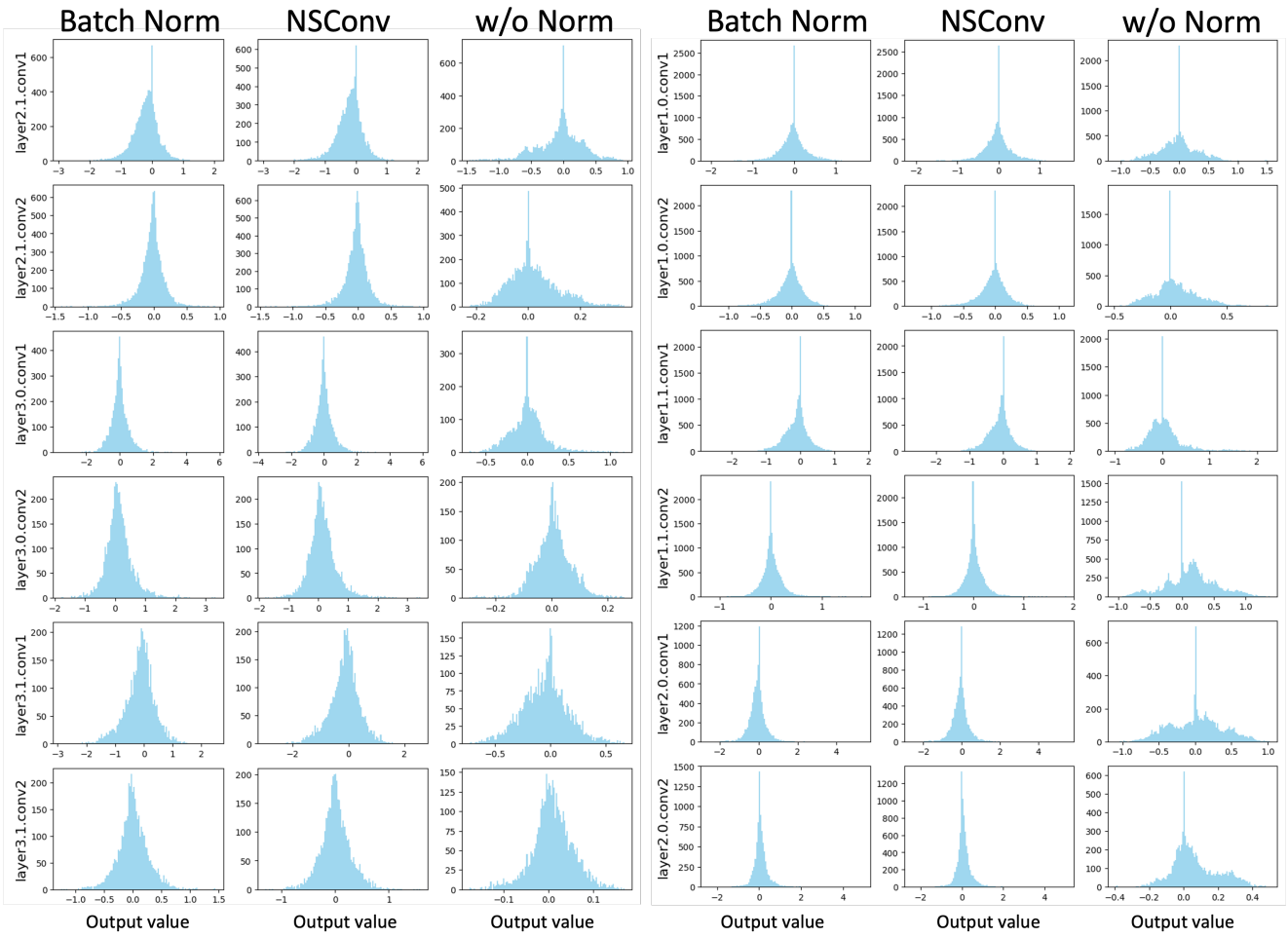

Figure 8. Distribution of output from all convolution layers in ResNet18 model using Batch Normalization layers (BatchNorm), without normalization layers (w/o Norm), and with Normalized Sparse Convolution (NSConv). The range of activation values exhibits a decrease, and the distribution becomes more centralized as the layer deepens. This observation aligns with Equations 11 and 12, which suggest that the mean and variance values will be scaled with $1/c_{in}$ and $1/c_{in}^2$, respectively.

7.2. The Memory Footprint of Training Models

We estimate the memory footprint for training to be a combination of parameters, activations, activation gradients, and parameter gradients. The memory for parameters is equal to the storage of parameters. We estimate the memory for activations by taking the maximum value of multiple measurements. For simplicity, we set the memory for gradients of activations to be equal to the memory for activations. We do not consider the memory for 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 dense and sparse activations are M_d^a and M_s^p, the training memory for each algorithm would be:

- **FedAVG.** This technique requires the training of a dense model; thus, the memory for the gradients of parameters is close to M_d^p. The memory footprint for training is approximately $2M_d^p + 2M_d^a$.
- **FL-PQSU.** This technique trains a static sparse model, so the memory for parameter gradients is close to M_s^p. The memory needed for training is approximately $2M_s^p + 2M_d^a$.
- **FedTiny and FedDST.** Since these methods only update the TopK gradients in memory to adjust the model structure, extra memory is used to store the top-ξ gradients and their indices. Therefore, the memory for the parameter gradients is approximate $M_s^p + M_\xi$, where M_ξ is the memory for the TopK gradients. Consequently, the total memory footprint is $2M_s^p + 2M_d^a + M_\xi$.
- **FedMef.** In comparison to FedTiny and FedDST, FedMef applies scaled activation pruning to activation, resulting in a cache memory of activation of M_d^a. However, the activation gradients are not pruned, leading to a total

memory footprint of $2M_s^p + M_s^a + M_d^a + M_\xi$.

7.3. Training FLOPs

Compared to other baselines, FedMef incurs minimal computational overhead. Firstly, in budget-aware extrusion, the computational overhead is attributed to the calculation of the regularization term $||\theta_{low}||$, with a complexity of $O(|\theta_{low}|) = O(|\theta|)$. Second, in Scaled Activation Pruning, the computational overhead arises from the normalization in Normalized Sparse Convolution and activation pruning. The normalization operation applies only to unpruned parameters, resulting in a computational complexity of $O((1 - s_m)|\theta|) = O(|\theta|)$. Additionally, the complexity associated with activation pruning is denoted as $O(|a| \log |a|)$, where $|a|$ represents the number of activation elements. The cumulative computational overhead is thus defined as $O(|\theta| + |a| \log |a|)$. These computational overheads are considered negligible compared to the intricate computations during training.

Training FLOPs comprise both forward pass FLOPs and backward pass FLOPs, where the total operations are tallied layer by layer. In the forward pass, layer activations are computed sequentially using previous activations and layer parameters. During the backward pass, each layer computes the activation gradients and the parameter gradients, assuming **twice** as many FLOPs in the backward pass as in the forward pass. FLOPs in batch normalization and loss calculation are omitted.

In detail, assuming that the inference FLOPs for dense and static sparse models are F_d and F_s, and the local iteration number is E, the maximum training FLOPs for each framework are as follows:

- **FedAVG.** Necessitates training a dense model, resulting in training FLOPs per round equal to $3F_dE$.
- **FedTiny and FedDST.** Utilizes RigL-based methods to update model architectures, requiring clients to calculate dense gradients in the last iteration. The maximum training FLOPs are $3F_s(E - 1) + F_s + 2F_d$.
- **FedMef.** Compared to FedTiny and FedDST, FedMef incurs a slight calculation overhead for BaE and SAP. Therefore, the maximum training FLOPs are $3(F_s + F_o)(E - 1) + (F_s + F_o) + 2F_d$, where F_o is the computing overhead of BaE and SAP. We estimate F_o as $F_o = 4(1 - s_m)n_\theta + n_a \log n_a$, where n_θ is the number of parameters θ and n_a is the number of activation elements. $4(1 - s_m)n_\theta$ represents the FLOPs of regularization and WSConv, while $n_a \log n_a$ denotes the FLOPs for activation pruning.

7.4. The Communication Cost

Regarding the communication costs, FedMef aligns with the communication cost of FedTiny [17]. In contrast to other baselines such as FedDST, wherein, for every ΔR rounds, clients are required to upload the TopK gradients to the server to support parameter growth, the number of TopK gradients ξ_t is aligned with the count of marked parameters θ_{low}, where $\xi_t = \zeta_t(1-s_m)n_\theta$ and $\zeta_t = 0.2(1+cos\frac{t\pi}{R_{stop}E})$ is the adjustment rate for the t-th iteration. Consequently, the upload overhead is minimal. Furthermore, there is no communication overhead for the model mask m during download because sparse storage formats, such as bitmap and coordinate offset, contain identical element position information. We omit other auxiliary data, such as the learning rate schedule.

Therefore, assuming that the storage for dense and sparse parameters is O_d and O_s, respectively, the data exchange per round is:

- **FedAVG.** The data exchange is $2O_d$, containing uploading and downloading dense parameters.
- **FedDST.** As mentioned above, the model mask does not require extra space to store, as the compressed sparse parameters already contain the mask information. Therefore, the data exchange per round is $2O_s$, including uploading and downloading sparse parameters.
- **FedTiny and FedMef.** Compared to FL-PQSU and FedDST, FedTiny and FedMef require uploading TopK gradients every ΔR rounds. Therefore, the maximum data exchange per round is $2O_s + O_\xi$, where O_ξ denotes the storage of the TopK gradients.

8. More Experiments

To showcase the efficiency of the proposed FedMef, we conducted experiments in various federated learning scenarios. Moreover, we also analyze the sensitivity to key hyperparameters in the proposed FedMef. Additionally, to demonstrate the efficacy of FedMef in various model architectures, we selected ResNet34 and ResNet50 for experimentation.

8.1. Impact of Local Epochs Number

Due to resource constraints and limited device battery life, the number of local training epochs is necessarily restricted. However, this constraint may impact the training of federated pruning frameworks and potentially undermine their performance. To assess this, we evaluate FedMef and other baseline frameworks on the CIFAR-10 dataset under varying local epoch numbers, employing the ResNet34 model.

As depicted in Table 3, our findings reveal that a smaller number of local epochs can affect the performance of FedAVG, and this impact extends to the federated pruning framework as well. Nevertheless, FedMef consistently outperforms other baselines. Notably, the performance of FedDST experiences a significant decline when the local epoch number is very low, underscoring the necessity of sufficient local training for FedDST before adjusting model architecture.

# Epochs	10	5	2
FedAVG	82.32%	79.87%	70.15%
FedDST	80.30%	76.04%	64.70%
FedTiny	80.41%	77.35%	70.01%
FedMef	**81.18%**	**78.46%**	**70.06%**

Table 3. Mean accuracy of FedMef and baseline frameworks on CIFAR-10 with various numbers of local epochs per round using ResNet34 model.

# Clients	10	5	3	1
FedAVG	78.25%	77.47%	74.94%	66.38%
FedDST	72.28%	73.21%	68.26%	56.05%
FedTiny	74.42%	74.52%	72.36%	**61.45%**
FedMef	**76.24%**	**76.33%**	**72.41%**	61.04%

Table 4. Mean accuracy of FedMef and baseline frameworks on CIFAR-10 with various numbers of selected clients per round using ResNet50 model.

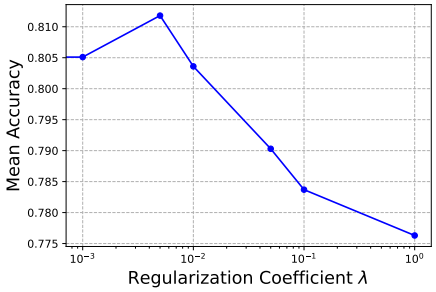

Figure 9. The mean accuracy of the proposed FedMef with various regularization coefficient λ

8.2. Impact of Selected Clients Number

Due to diverse network conditions on devices, the number of clients participating in each round is limited. However, this constraint, while improving the negative effect of data heterogeneity, can slow down the convergence speed and affect final performance. In our evaluation, we evaluate the proposed FedMef and other baseline frameworks with various numbers of selected clients per round, utilizing the ResNet50 model.

As presented in Table 4, FedMef consistently outperforms other baselines in most cases. Notably, the performance of FedDST decreases significantly compared to our FedMef and FedTiny, underscoring the necessity of sufficient local training for FedDST.

8.3. Impact of Regularization Coefficient λ

We assess the sensitivity of the regularization coefficient (λ) in the proposed budget-aware extrusion (BaE). Different coefficients of λ are set in FedMef and experiments are conducted on the CIFAR-10 dataset using the ResNet34 model.

Acc(Memory)	FedMef($s_{tm} = 0.8$)	FedAVG + SAP	FedAVG
MobileNetV2	65.50%(84.94 MB)	64.04%(101.28 MB)	64.28%(148.63 MB)
ResNet18	81.73%(45.91 MB)	81.32%(111.51 MB)	81.15%(120.74MB)

Table 5. Accuracy and memory footprint on the CIFAR-10 dataset

As illustrated in Figure 9, the accuracy of FedMef initially increases and then decreases sharply as the coefficient λ increases. The initial increase demonstrates the effectiveness of budget-aware extrusion, while the subsequent decrease is attributed to large λ values that rapidly zero out the parameters, resulting in an excessively sparse model.

8.4. Empirical comparison with dense model

To further demonstrate the effectiveness of BaE and SAP, we conducted experiments on both dense and sparse models on the CIFAR-10 dataset. The results, as illustrated in Figure 5, indicate that SAP maintains the performance of dense models effectively, while reducing the memory footprint of activation. Furthermore, FedMef with BaE outperforms its counterpart without BaE (FedAVG+SAP), underscoring BaE's contribution to improving performance.

9. More Detail and Information

9.1. NSConv v.s. BN

NSConv is more suitable for CNNs, which are popular on edge devices. It excels with sparse weights, as it may introduce more computational overhead on dense weights. NSConv outperforms BN when the batch size is small, as shown in Table 5 right (FedMef v.s. FedMef w/o SAP). This is important for low-memory devices that can only train with a small batch size. NSConv matches BN performance when batch size is large, as shown in Figure 8 .

9.2. Structured pruning v.s. Unstructured pruning

We use unstructured pruning(pruning parameters) instead of structured pruning (pruning filters) as structured pruning often suffers from a serious performance drop when the sparsity is higher than 10% [42] and has a limited impact on memory reduction. In contrast, our proposed unstructured pruning method can achieve 80%+ sparsity while maintaining accuracy, as shown in Table 1.

9.3. Convergence of Model Structure

We adopt the sparse dynamic pruning technique [10, 11, 32, 35] which starts training from a random sparse network and eventually evolves into the final sparse model structure with accuracy comparable to starting from a dense model. Previous research [35] empirically suggests that the structure converges faster than the parameters, but the theoretical guarantees remain unexplored, which we plan to investigate in the future.