A Pseudocodes

A.1 Global Contribution Partial Updating

The magnitude pruning method prunes (*i.e.*, set as zero) weights with the lowest magnitudes in a model, which often yields a good trade-off between the model accuracy and the number of zero's weights [29]. We adapt the magnitude pruning proposed in [29] to prune the incremental weights $\delta \boldsymbol{w}^{\mathrm{f}}$. Specially, the elements with the smallest absolute values in $\delta \boldsymbol{w}^{\mathrm{f}}$ are set to zero (also rewinding), while the remaining weights are further sparsely fine-tuned with the same learning rate schedule as training $\boldsymbol{w}^{\mathrm{f}}$.

Algorithm 2: Global Contribution Partial Updating (Prune Incremental Weights)

Input: weights $\boldsymbol{w}$, updating ratio k, learning rate $\{\alpha^q\}_{q=1}^{Q}$ in Q iterations
Output: weights $\widetilde{\boldsymbol{w}}$
`/* The first step: full updating and rewinding */`
Initiate $\boldsymbol{w}^0 = \boldsymbol{w}$;
for $q \leftarrow 1$ **to** Q **do**
  Compute the loss gradient $\boldsymbol{g}(\boldsymbol{w}^{q-1}) = \partial \ell(\boldsymbol{w}^{q-1}) / \partial \boldsymbol{w}^{q-1}$;
  Compute the optimization step with learning rate α^q as $\Delta \boldsymbol{w}^q$;
  Update $\boldsymbol{w}^q = \boldsymbol{w}^{q-1} + \Delta \boldsymbol{w}^q$;
Set $\boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^Q$ and get $\delta \boldsymbol{w}^{\mathrm{f}} = \boldsymbol{w}^{\mathrm{f}} - \boldsymbol{w}$;
Compute $\boldsymbol{c}^{\mathrm{global}} = \delta \boldsymbol{w}^{\mathrm{f}} \odot \delta \boldsymbol{w}^{\mathrm{f}}$ and sort in descending order;
Create a binary mask $\boldsymbol{m}$ with 1 for the Top-$(k \cdot I)$ indices, 0 for others;
`/* The second step: sparse fine-tuning */`
Initiate $\widetilde{\delta \boldsymbol{w}} = \delta \boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta \boldsymbol{w}}$;
for $q \leftarrow 1$ **to** Q **do**
  Compute the optimization step on $\widetilde{\boldsymbol{w}}$ with learning rate α^q as $\Delta \widetilde{\boldsymbol{w}}^q$;
  Update $\widetilde{\delta \boldsymbol{w}} = \widetilde{\delta \boldsymbol{w}} + \Delta \widetilde{\boldsymbol{w}}^q \odot \boldsymbol{m}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \widetilde{\delta \boldsymbol{w}}$;

In comparison to traditional pruning on weights, pruning on incremental weights has a different start point. Traditional pruning on weights first trains randomly initialized weights (a zero-initialized model cannot be trained due to the symmetry), and then prunes the weights with the smallest magnitudes. However, the increment of weights $\delta \boldsymbol{w}^{\mathrm{f}}$ is initialized with zero in Alg. 2, since the first step starts from $\boldsymbol{w}$. This implies that pruning $\delta \boldsymbol{w}^{\mathrm{f}}$ has the same functionality as rewinding these weights to their initial values in $\boldsymbol{w}$.

B Complexity Analysis

Algorithm 1: Deep Partial Updating. Recall that the total number of weights vector is denoted as I. In Q optimization iterations during the first step, Alg. 1

introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. The rest of the first step takes a time complexity of $O(I \cdot \log(I))$ and a space complexity of $O(I)$ (*e.g.*, using heap sort or quick sort). In Q optimization iterations during the second step, Alg. 1 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. Thus, a total extra time complexity is $O(2QI + I \cdot \log(I))$ and a total extra space complexity is $O(I)$.

Algorithm 2: Global Contribution Partial Updating. Recall that the total number of weights vector is denoted as I. In Q optimization iterations during the first step, Alg. 2 does not introduce extra time complexity or extra space complexity related to the original optimizer. The rest of the first step takes a time complexity of $O(I \cdot \log(I))$ and a space complexity of $O(I)$ (*e.g.*, using heap sort or quick sort). In Q optimization iterations during the second step, Alg. 2 introduces an extra time complexity of $O(QI)$, and an extra space complexity of $O(I)$ related to the original optimizer. Thus, a total extra time complexity is $O(QI + I \cdot \log(I))$ and a total extra space complexity is $O(I)$.

C Implementation Details

C.1 MLP on MNIST

The MNIST dataset [16] consists of 28×28 gray scale images in 10 digit classes. It contains a training dataset with 60000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 GeForce RTX 3090 GPU. We use Adam variant of SGD as the optimizer, and use all default parameters provided by Pytorch. The number of training epochs is chosen as 60 at each round. The initial learning rate is 0.005, and it decays with a factor of 0.1 every 20 epochs. The used MLP contains two hidden layers, and each hidden layer contains 512 hidden units. The input is a 784-dim tensor consisting of all pixel values in each image. We use ReLU as the activation function, and use a softmax function as the non-linearity of the last layer (*i.e.*, the output layer) in the entire paper. All weights in MLP need around 2.67MB. Each data sample needs 0.784KB. The size of MLP equals around 3400 data samples. The used MLP architecture is presented as, 2×512FC - 10SVM.

C.2 VGGNet on CIFAR10

The CIFAR10 dataset [15] consists of 32×32 color images in 10 object classes. It contains a training dataset with 50000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 GeForce RTX 3090 GPU. We use Adam variant of SGD as the optimizer, and use all

default parameters provided by Pytorch. The number of training epochs is chosen as 60 at each round. The initial learning rate is 0.005, and it decays with a factor of 0.2 every 20 epochs. The used VGGNet is widely adopted in many previous compression works [5,28,26], which is a modified version of the original VGG [33]. All weights in VGGNet need around 56.09MB. Each data sample needs 3.072KB. The size of VGGNet equals around 18200 data samples. The used VGGNet architecture is presented as, 2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM.

C.3 ResNet56 on CIFAR100

Similar as CIFAR10, the CIFAR100 dataset [15] consists of 32×32 color images in 100 object classes. It contains a training dataset with 50000 data samples, and a test dataset with 10000 data samples. We use the original training dataset for training; and randomly select 3000 samples in the original test dataset for validation, and the rest 7000 samples for testing. We use a mini-batch with size of 128 training on 1 GeForce RTX 3090 GPU. We use Nesterov SGD with weight decay 0.0001 as the optimizer, and use all default parameters provided by Pytorch. The number of training epochs is chosen as 100 at each round. The initial learning rate is 0.1, and it decays with the cosine annealing schedule. The ResNet56 used in our experiments is proposed in [10]. All weights in ResNet56 need around 3.44MB. Each data sample needs 3.072KB. The size of ResNet56 equals around 1100 data samples.

C.4 MobileNetV1 on ImageNet

The ImageNet dataset [30] consists of high-resolution color images in 1000 object classes. It contains a training dataset with 1.28 million data samples, and a validation dataset with 50000 data samples. Following the commonly used preprocessing [25], each sample (single image) is randomly resized and cropped into a 224×224 color image. We use the original training dataset for training; and randomly select 15000 samples in the original validation dataset for validation, and the rest 35000 samples for testing. We use a mini-batch with size of 1024 training on 4 GeForce RTX 3090 GPUs. We use Nesterov SGD with weight decay 0.0001 as the optimizer, and use all default parameters provided by Pytorch. The number of training epochs is chosen as 150 at each round. The initial learning rate is 0.5, and it decays with the cosine annealing schedule. The MobileNetV1 used in our experiments is proposed in [12]. All weights in MobileNetV1 need around 16.93MB. Each data sample needs 150.528KB. The size of MobileNetV1 equals around 340 data samples.

D Full Updating

Settings. In this experiment, we compare full updating with different initialization at each round to confirm the best-performed full updating baseline. The

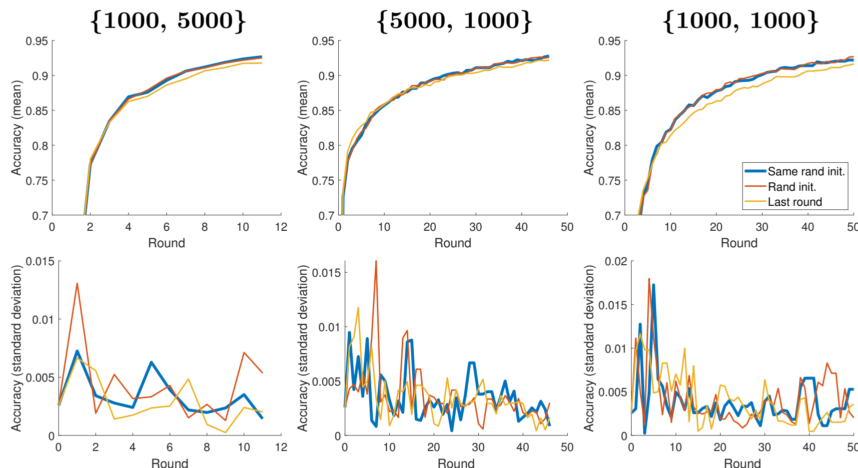

Fig. 4: Comparing full updating methods with different initialization at each round.

compared full updating methods include, (i) the model is trained from different random initialization at each round; (ii) the model is trained from a same random initialization at each round, *i.e.*, with the same random seed; (iii) the model is trained from the weights $\boldsymbol{w}^{r-1}$ of the last round at each round. The experiments are conducted on VGGNet using CIFAR10 dataset with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$. Each experiment runs for three times using random data samples.

Results. We report the mean and the standard deviation of test accuracy (over three runs) under different initialization in Fig. 4. The results show that training from a same random initialization yields a similar accuracy level while sometimes also a lower variance, as training from different random initialization at each round. In comparison to training from scratch (*i.e.*, random initialization), training from $\boldsymbol{w}^{r-1}$ may yield a higher accuracy in the first few rounds; yet training from scratch can always outperform after a large number of rounds. Thus, in this paper, we adopt training from a same random initialization at each round, *i.e.*, (ii), as the baseline of full updating.

E Number of Rounds for Re-Initialization

Settings. In these experiments, we re-initialize the model every n rounds under different partial updating settings to determine a heuristic rule to set the number of rounds for re-initialization. We conduct experiments on VGGNet using CIFAR10 dataset, with different amounts of training samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Every n rounds, the model is (re-)initialized again from a same random model (as mentioned in D), then partially updated in the

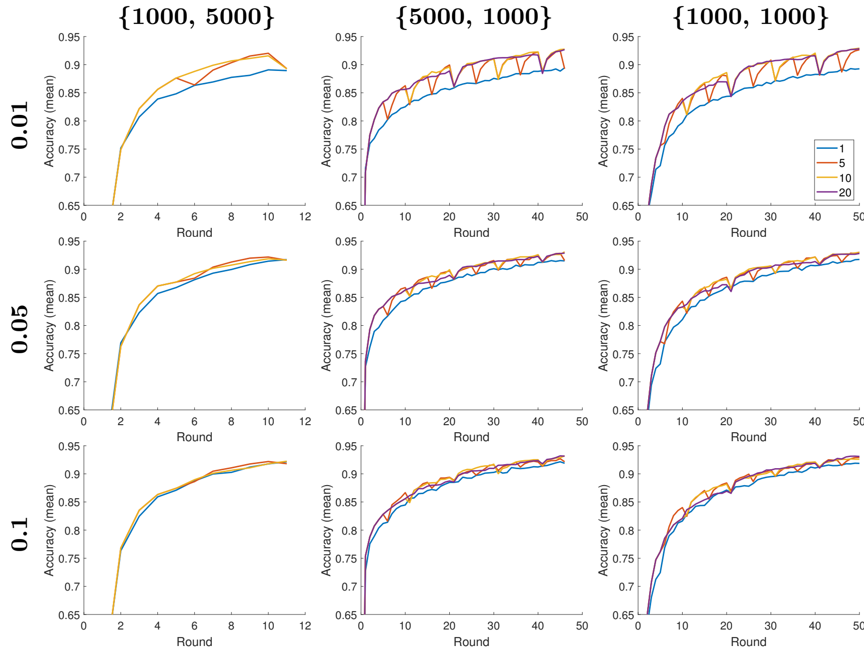

Fig. 5: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

next n rounds with Alg. 1. We choose $n = 1, 5, 10, 20$. Specially, $n = 1$ means that the model is partially updated from the same random model every round, *i.e.*, without reusing the learned knowledge at all. Each experiment runs three times using random data samples.

Results. We plot the mean test accuracy along rounds in Fig. 5. By comparing $n = 1$ with other settings, we can conclude that within a certain number of rounds, the current deployed model $\boldsymbol{w}^{r-1}$ (*i.e.*, the model from the last round) is a better starting point for Alg. 1 than a randomly initialized model. In other word, partially updating from the last round may yield a higher accuracy than partially updating from a random model with the same training effort. This is straightforward, since such a model is already pretrained on a subset of the currently available data samples, and the previous learned knowledge could help the new training process. Since all newly collected samples are continuously stored in the server, complete information about the past data samples is available. This also makes our setting different from continual learning setting, which aims at avoiding catastrophic forgetting without accessing (at least not all) old data.

Each time the model is re-initialized, the new partially updated model might suffer from an accuracy drop in a few rounds. Although this accuracy drop may be relieved if we carefully tune the partial updating training scheme every time,

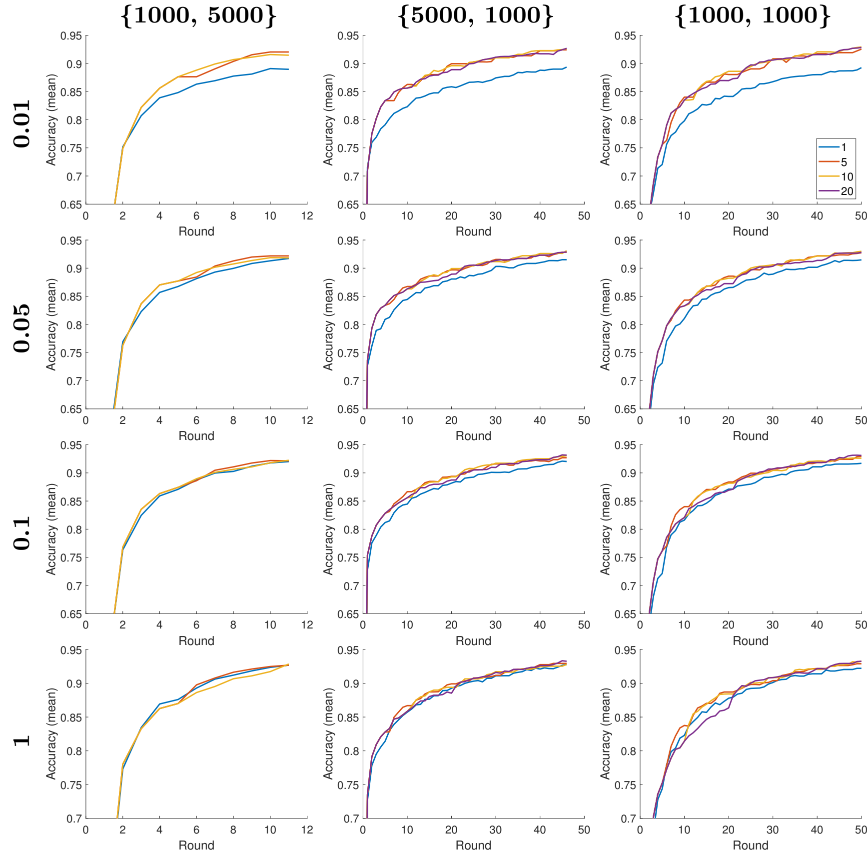

Fig. 6: Comparison w.r.t. the mean accuracy when DPU is re-initialized every n rounds ($n = 1, 5, 10, 20$) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$ and full updating $k = 1$) settings.

this is not feasible regarding a large number of updating rounds. However, we can simply avoid such an accuracy drop by not updating the model if the validation accuracy does not increase compared to the last round (as discussed in Sec. 5). Note that in this situation, the partially updated weights (as well as the random seed for re-initialization) still need to be sent to the edge devices, since this is an on-going training process. After implementing the above strategy, we plot the mean accuracy in Fig. 6. In addition, we also add the related results of full updating in Fig. 6, where the model is fully updated and is re-initialized every n rounds from a same random model. Note that full updating with re-initialization every round ($n = 1$) is exactly the same as "same rand init." in Fig. 4 in D. From Fig. 6, we can conclude that the model needs to be re-initialized more frequently in the first several rounds than in the following rounds to achieve a higher accuracy level. The model also needs to be re-initialized more frequently with a large partial updating ratio k. Particularly, the ratio between the number of current data samples and the number of following collected data samples has a larger impact than the updating ratio.

Thus, we propose to re-initialize the model as long as the number of total newly collected data samples exceeds the number of samples when the model is re-initialized last time. For example, assume that at round r the model is randomly (re-)initialized and partially updated from the random model on dataset $\mathcal{D}^r$. Then, the model will be re-initialized at round $r + n$, if $|\mathcal{D}^{r+n}| > 2 \cdot |\mathcal{D}^r|$.

F Additional Multi-Round Updating Results

F.1 Experiments on Total Communication Cost Reduction

Settings. In this experiment, we show the advantages of DPU in terms of the total communication cost reduction, as DPU has no impact on the edge-to-server communication which may involve sending new data samples collected on edge nodes. The total communication cost includes both the edge-to-server communication and the server-to-edge communication. Here we assume that all samples in $\delta\mathcal{D}^r$ are collected by N edge nodes during all rounds and sent to the server on a per-round basis. In other words, the first stage (see in Sec. 1) is anyway necessary for sending new training data to the server. For clarity, let S_d denote the data size of each training sample. During round r, we define the per-node total communication cost under DPU as $S_d \cdot |\delta\mathcal{D}^r|/N + (S_w \cdot k \cdot I + S_x(k) \cdot I)$. Similarly, the per-node total communication cost under full updating is defined as $S_d \cdot |\delta\mathcal{D}^r|/N + S_w \cdot I$.

In order to simplify the demonstration, we consider the scenario where N nodes send a certain amount of data samples to the server in $R - 1$ rounds, namely $\sum_{r=2}^{R} |\delta\mathcal{D}^r|$ (see Sec. 4.2). Thus, the average data size transmitted from each node to the server in all rounds is $\sum_{r=2}^{R} S_d \cdot |\delta\mathcal{D}^r|/N$. A larger N implies a fewer amount of transmitted data from each node to the server.

Results. We report the ratio of the total communication cost over all rounds required by DPU related to full updating, when DPU achieves a similar accuracy

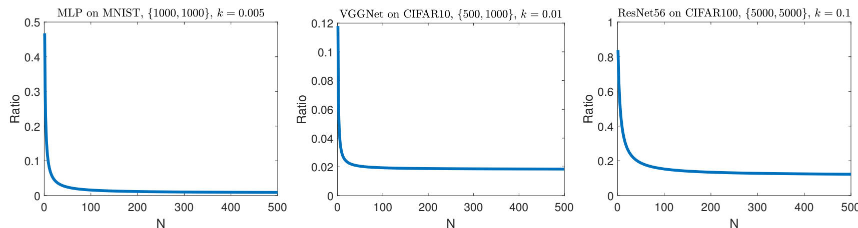

Fig. 7: The ratio, between the total communication cost (over all rounds) under DPU and that under full updating, varies with the number of edge nodes N.

level as full updating (corresponding to three evaluations in Fig. 2). The ratio clearly depends on $\sum_{r=2}^{R} S_d \cdot |\delta\mathcal{D}^r|/N$, *i.e.*, the number of nodes N. The relation between the ratio and N is plotted in Fig. 7.

DPU can reduce up to 88.2% of the total communication cost even for a single node. Single node corresponds to the largest data size during edge-to-serve transmission per node, *i.e.*, the worst case. Moreover, DPU tends to be more beneficial when the size of data transmitted by each node to the server becomes smaller. This is intuitive because in this case the server-to-edge communication (thus the reduction due to DPU) dominants in the entire communication.

F.2 Impact due to Varying Number of Data Samples and Updating Ratios

Settings. In this section, we show that DPU outperforms other baselines under varying number of training samples and updating ratios in multi-round updating. We also conduct ablations concerning the re-initialization of weights discussed in Sec. 4.2. We implement DPU with and without re-initialization, GCPU with and without re-initialization, RPU, and Pruning [29] (see more details in Sec. 5.1) on VGGNet using CIFAR10 dataset. We compare these methods with different amounts of samples $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and different updating ratios k. Each experiment runs three times using random data samples.

Results. We compare the difference between the accuracy under each partial updating method and that under full updating. The mean accuracy difference (over three runs) is plotted in Fig. 8. As seen in Fig. 8, DPU (with re-initialization) always achieves the highest accuracy. DPU also significantly outperforms the pruning method, especially under a small updating ratio. Note that we preferred a smaller updating ratio in our context because it explores the limits of the approach and it indicates that we can improve the deployed model more frequently with the same accumulated server-to-edge communication cost. In addition, we also plot the mean and standard deviation of the absolute accuracy of these methods (including full updating) in Fig. 9 and Fig. 10, respectively. The dashed curves and the solid curves with the same color in these figures can be viewed as the ablation study of our re-initialization scheme. Particularly given a large number

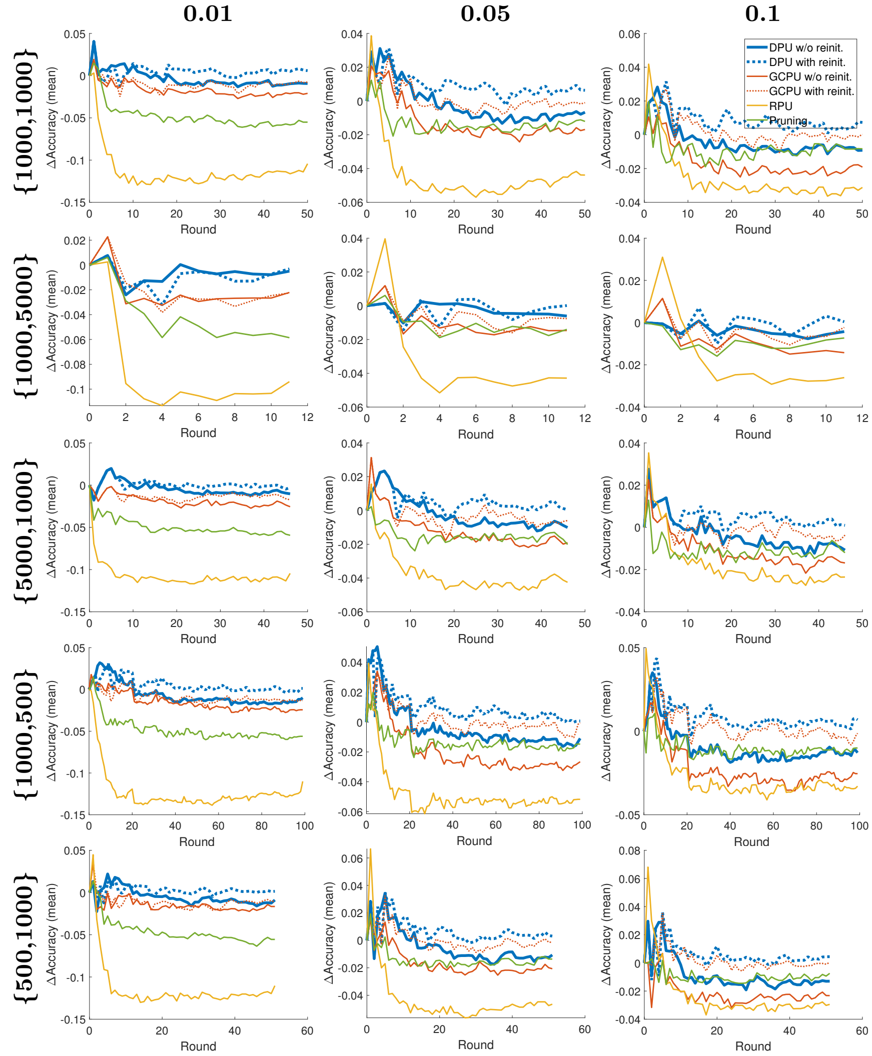

Fig. 8: Comparison w.r.t. the mean accuracy difference (full updating as the reference) under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

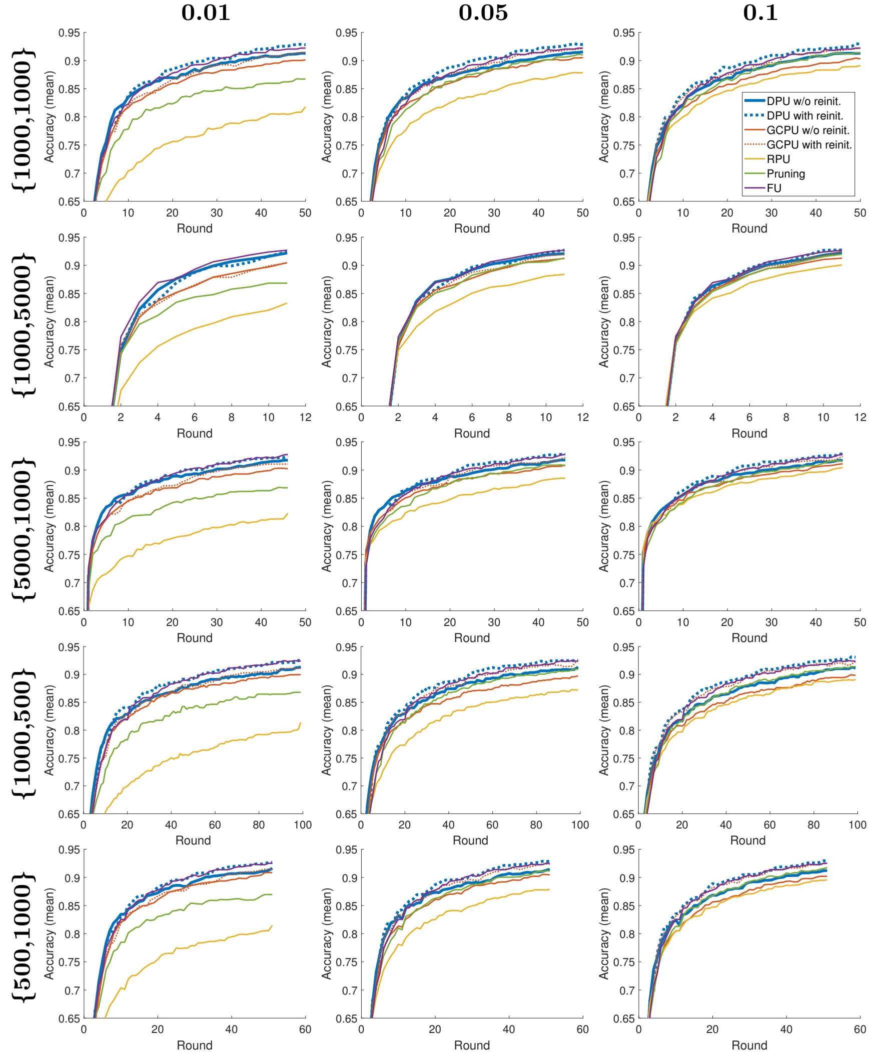

Fig. 9: Comparison w.r.t. the mean accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

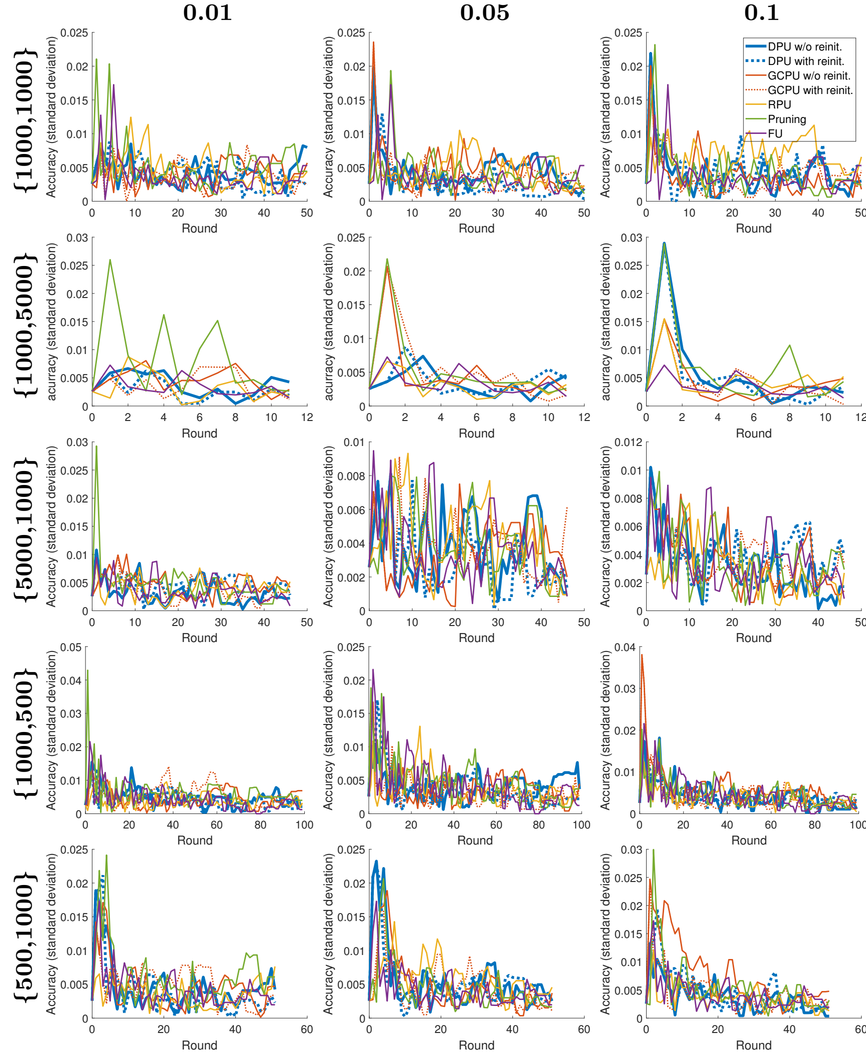

Fig. 10: Comparison w.r.t. the standard deviation of accuracy under different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ and updating ratio ($k = 0.01, 0.05, 0.1$) settings.

of rounds, it is critical to re-initialize the start point $\boldsymbol{w}^{r-1}$ after several rounds (as discussed in Sec. 4.2).

G Impacts from Global/Local Contributions

G.1 Ablation Studies of Rewinding Metrics

Settings. We conduct a set of ablation experiments regarding different rewinding metrics discussed in Sec. 4.1. We compare the influence of the local and global contributions as well as their combination, in terms of the training loss after the rewinding and the final test accuracy. We conduct single-round updating on VGGNet. The initial model are fully trained on a randomly selected dataset of 10^3 samples. After adding 10^3 new randomly selected samples, we conduct the first step of our approach (see Alg. 1) with all three rewinding metrics, *i.e.*, the global contribution, the local contribution, and the combined contribution. Accordingly, the second step (sparse fine-tuning) is executed. The experiment is executed over five runs with different random seeds.

Results. The training loss after rewinding (*i.e.*, $\ell(\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m})$) and the final test accuracy after sparse fine-tuning (*i.e.*, at $\widetilde{\boldsymbol{w}}$) are reported in Tab. 3. We use the form of mean $\pm$ standard deviation. As seen in the table, the combined contribution always yields a lower or similar training loss after rewinding compared to the other two metrics. The smaller deviation also indicates that adopting the combined contribution yields more robust results. This demonstrates the effectiveness of our proposed metric, *i.e.*, the combined contribution to the analytical upper bound on loss reduction. Rewinding with the combined contribution also acquires a higher final accuracy, which in turn verifies the hypothesis we made for partial updating, a weight shall be updated only if it has a large contribution to the loss reduction.

Table 3: Comparing training loss after rewinding and the final test accuracy under different metrics.

k	Training loss at $\boldsymbol{w} + \delta\boldsymbol{w}^{\mathrm{f}} \odot \boldsymbol{m}$ (Test accuracy at $\widetilde{\boldsymbol{w}}$)		
	Global	Local	Combined
0.01	3.04 ± 0.07 (55.0 ± 0.1%)	**2.59** ± 0.08 (55.6 ± 0.1%)	2.66 ± 0.09 (**56.5** ± 0.0%)
0.05	2.51 ± 0.06 (57.3 ± 0.2%)	1.80 ± 0.10 (57.8 ± 0.1%)	**1.67** ± 0.06 (**58.2** ± 0.1%)
0.1	2.03 ± 0.05 (58.3 ± 0.0%)	1.34 ± 0.08 (59.0 ± 0.1%)	**0.99** ± 0.03 (**59.0** ± 0.1%)
0.2	1.20 ± 0.05 (59.0 ± 0.1%)	0.74 ± 0.03 (59.6 ± 0.2%)	**0.42** ± 0.01 (**60.1** ± 0.2%)

G.2 Balancing between Global and Local Contributions

Settings. In Eq.(10), the combined contribution is calculated by adding both normalized contributions together. However, both normalized contributions may

have different importance when determining the critical weights. In order to investigate which one plays a more essential role in the combined contribution, we introduce another hyper-parameter λ to tune the proportion of both normalized contributions as

$$\boldsymbol{c}_\lambda = \lambda \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{global}}} \boldsymbol{c}^{\mathrm{global}} + (1-\lambda) \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{local}}} \boldsymbol{c}^{\mathrm{local}} \tag{11}$$

Note that the combined contribution $\boldsymbol{c}$ used in the previous experiments is the same as $\boldsymbol{c}_\lambda$ when $\lambda = 0.5$, since only the order matters when determining the critical weights. We implement partial updating methods with the rewinding metric $\boldsymbol{c}_\lambda$ under different values of λ. We compare these methods under updating ratios $k = 0.01, 0.05, 0.1$ and different $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\}$ settings on VGGNet using CIFAR10 dataset, and with the re-initialization scheme described in Sec. 4.2. Each experiment runs three times using random data samples.

Results. To clearly illustrate the impact of λ, we compare the difference between the accuracy under partial updating methods with various λ and that under full updating. The mean accuracy difference (over three runs) are plotted in Fig. 11. As seen in Fig. 11, $\lambda = 0.5$ always obtains the best performance in general, especially when the updating ratio is small. Thus, in the following experiments, we fix this hyper-parameter λ as 0.5. In other words, the combined contribution is chosen as

$$\boldsymbol{c}_\lambda(\lambda = 0.5) = 0.5 \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{global}}} \boldsymbol{c}^{\mathrm{global}} + 0.5 \cdot \frac{1}{\mathbf{1}^{\mathrm{T}} \cdot \boldsymbol{c}^{\mathrm{local}}} \boldsymbol{c}^{\mathrm{local}} \tag{12}$$

which has exactly the same functionality as Eq.(10). Note that it may be possible to manually find another hyper-parameter λ that achieves better performance in certain cases. However, setting λ as 0.5 already yields a satisfactory performance, and can avoid meticulous and computationally expensive hyper-parameter tuning in a large number of updating rounds.

G.3 Number of Updated Weights across Layers under Different Rewinding Metrics

Settings. To further study the impact of adopting different rewinding metrics, we show the distribution of updated weights across layers in this section. We implement partial updating methods with three rewinding metrics (*i.e.*, the global contribution, the local contribution, and the combined contribution, see in Sec. 4.1) on VGGNet using CIFAR10 dataset. We compare these methods with different updating ratios k under $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}$. All methods start from the same randomly initialized model, and are re-initialized with this random model according to the proposed scheme in Sec. 4.2. To study the distribution of updated weights along all rounds, we let the model partially updated in every round even if the model accuracy may degrade for a few rounds due to the re-initialization.

Results. We plot the number of updated weights across all layers along rounds, under updating ratio $k = 0.01, 0.05, 0.1$ in Fig. 12, Fig. 13, and Fig. 14, respectively.

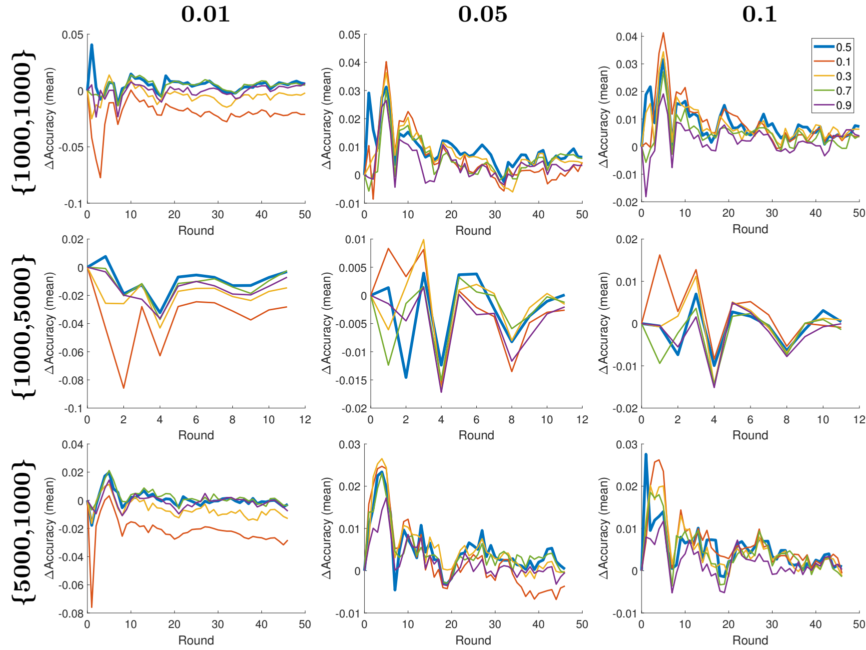

Fig. 11: Comparison w.r.t. the mean accuracy difference (full updating as the reference) under $\lambda = 0.5, 0.1, 0.3, 0.7, 0.9$. The chosen settings are updating ratios $k = 0.01, 0.05, 0.1$, $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}, \{1000, 5000\}, \{5000, 1000\}$.

We also plot the corresponding test accuracy along rounds in Fig. 15. Generally, the metric of local contribution updates more weights in the first several layers and the last layer while with a large variance along rounds. On the contrary, global contribution selects more weights in the last several layers (until the penultimate layer) to update. Combined contribution (the sum of normalized local/global contribution) achieves a more robust and balanced distribution of updated weights across layers than other contributions. It also results in the highest accuracy level especially under a small updating ratio. Intuitively, local contribution can better identify critical weights w.r.t. the loss during training, while global contribution may be more robust for a highly non-convex loss landscape. Both metrics may be necessary when selecting weights to rewind. Note that the proposed combined contribution is not the simple averaging of both local and global contribution. For example, in "layer 6" of Fig. 14, the number of updated weights by combined contribution already exceeds the other two metrics.

H Quantizing and Encoding

Settings. The updates could also be compressed through quantization and/or encoding to reduce the communication cost. In this set of experiments, we show these compression techniques (*i.e.*, quantization and encoding) are orthogonal to our DPU. [9] proposed that certain types of quantization and encoding could be applied in addition to pruning without hurting the accuracy. Following the compression pipeline in [9], the resulted sparse updating from our DPU could also be further quantized and Huffman-encoded. The overall compression pipeline *in each round* is summarized as follows, (*i*) a partial updating (also sparse updating) is generated from our DPU; (*ii*) these updates are quantized into 8-bit for each layer, *i.e.*, each layer's non-zero values share 256 centroids; (*iii*) the quantized updates are Huffman-encoded; (*iv*) the server sends the encoded updates, the code books (for Huffman-encoding and quantization), as well as the indices to edge devices.

We implement DPU (with re-initialization see in Sec. 4.2), DPU+Q+E, pruning (a state-of-the-art pruning method proposed in [29] see in Sec. 5.2), and pruning+Q+E, to verify that applying Q+E in addition does not bring extra accuracy loss. Here, Q stands for the quantization step in (*ii*), and E stands for the encoding step in (*iii*). We test on VGGNet using CIFAR10 dataset under $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}$, $\{1000, 5000\}$, and $\{5000, 1000\}$. Note that the updating ratio k is set to 0.01, also the most critical case. Each experiment runs three times using random data samples.

Results. We plot the mean and standard deviation of test accuracy (over three runs) of these methods in Fig. 16. In addition, we also add the baseline of full updating (FU) in the figures for comparison. The dashed curves and the solid curves with the same color can be viewed as the ablations of with/without quantization and Huffman-encoding, respectively. The results reveal that applying these quantization and encoding techniques does not bring performance degradation for both pruning methods and our deep partial updating schemes. Therefore, the

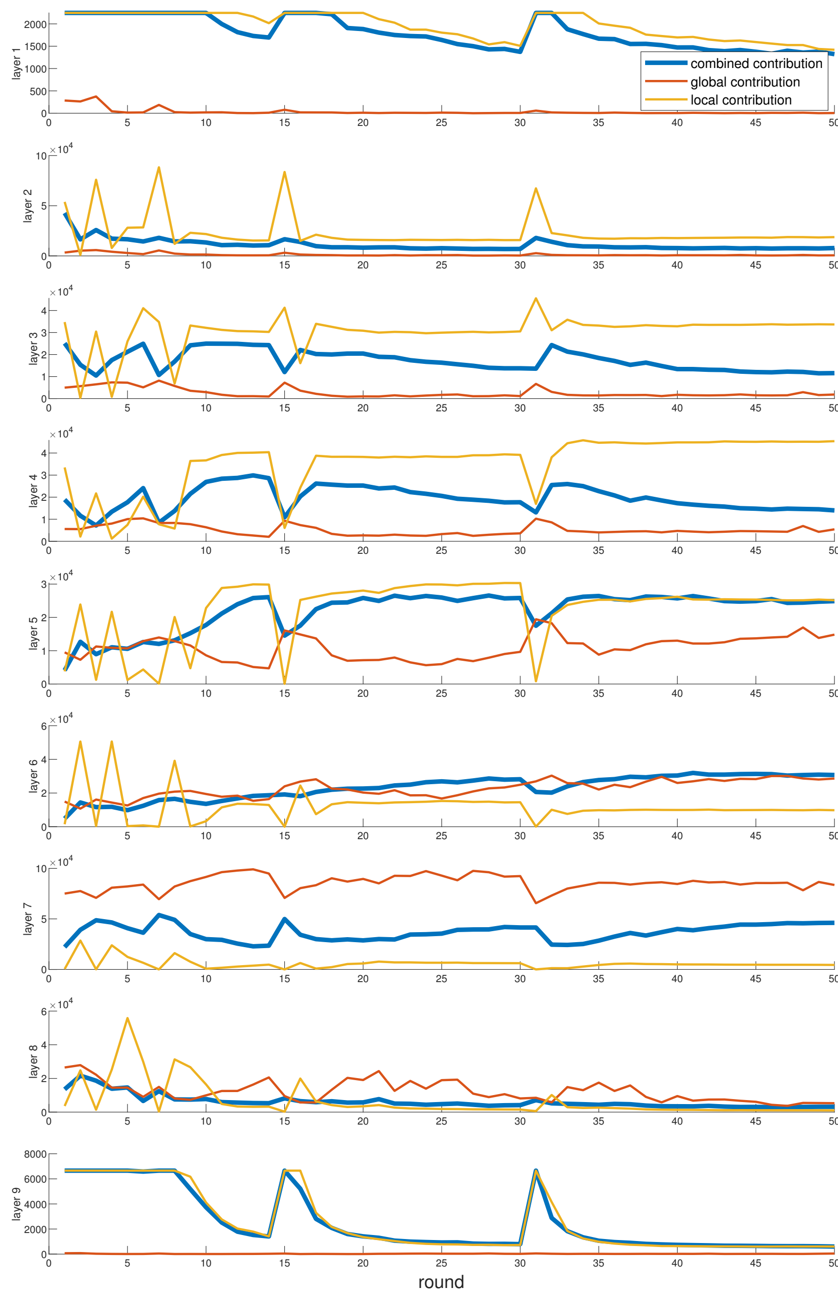

Fig. 12: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.01$).

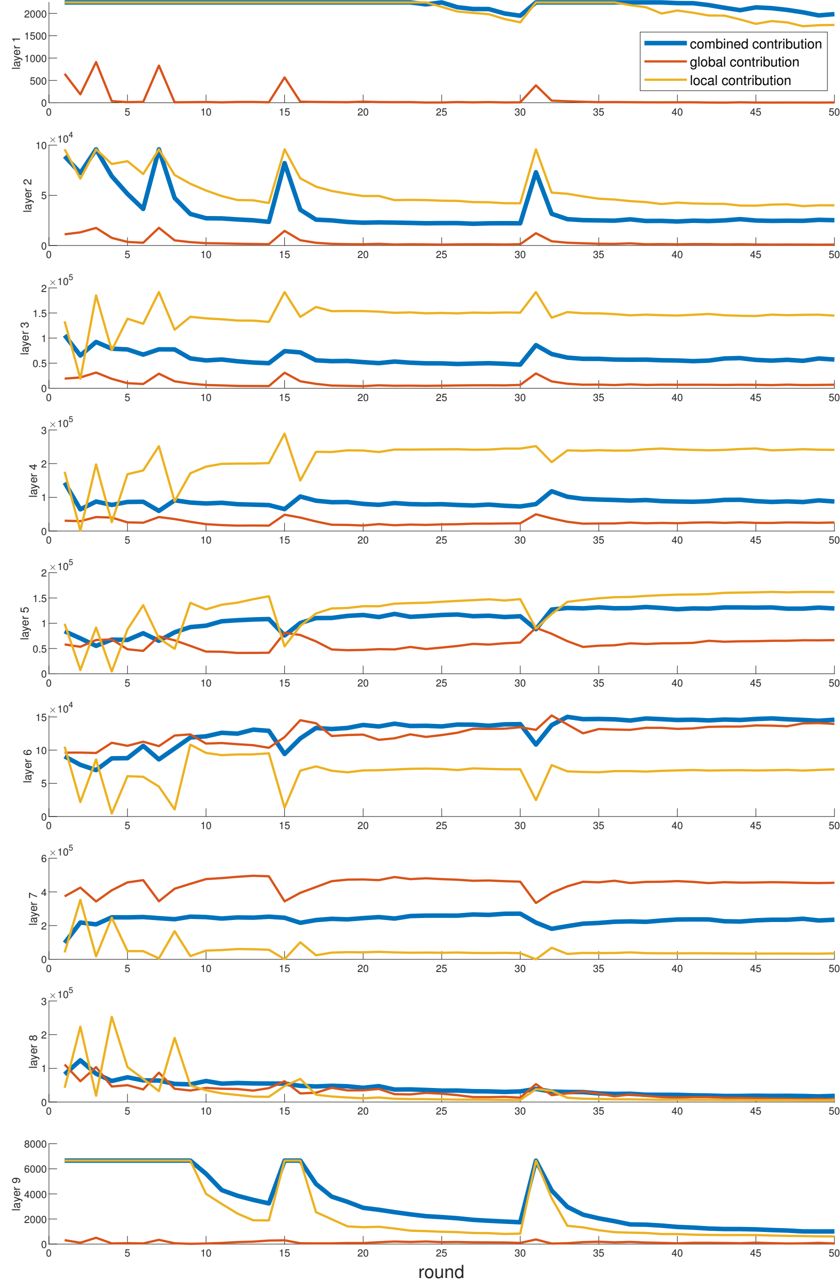

Fig. 13: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.05$).

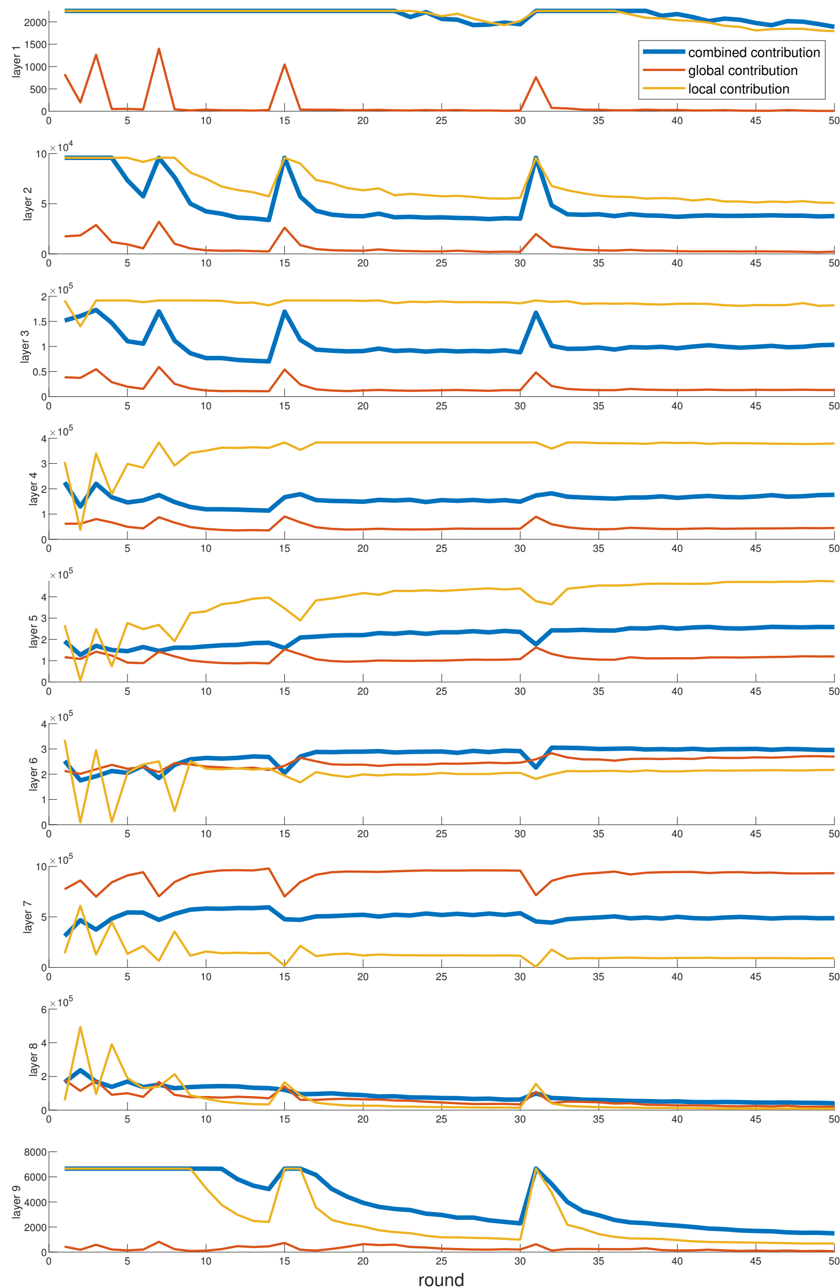

Fig. 14: Number of updated weights across all layers (VGGNet) when adopting different rewinding metrics (updating ratio $k = 0.1$).

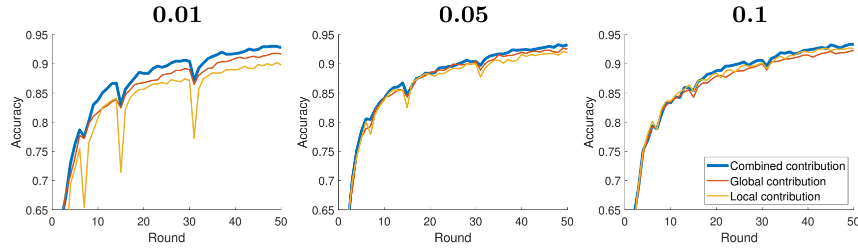

Fig. 15: The test accuracy of partial updating methods with different rewinding metrics (updating ratio $k = 0.01, 0.05, 0.1$).

size of transmitted data could be further reduced by quantizing and/or encoding the partial updates resulted from DPU. We report the ratio between the data size of server-to-edge transmission under these above methods and that under full updating in Tab. 4. Note that the reported ratios are the mean values averaged over all settings in Fig. 16. Particularly, in comparison to full updating, our DPU+Q+E can reduce the size of transmitted data by 145×, *i.e.*, 99.31%, while achieving a similar accuracy level.

Table 4: The ratio of communication cost (server-to-edge) over all rounds related to full updating.

Method	DPU	DPU+Q+E	Pruning	Pruning+Q+E
Ratio	0.0177	0.0069	0.0174	0.0068

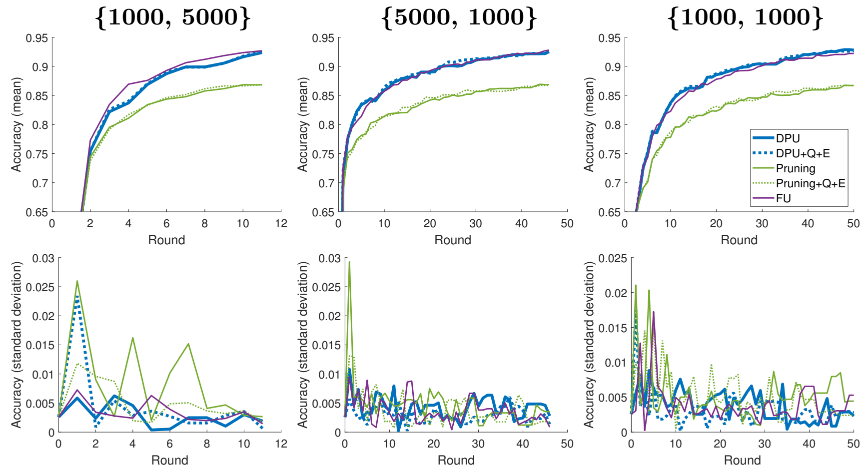

Fig. 16: Verifying the orthogonality between DPU (and pruning) and other compression techniques, namely quantization (Q) and Huffman-encoding (E). The chosen settings are updating ratio $k = 0.01$, $\{|\mathcal{D}^1|, |\delta\mathcal{D}^r|\} = \{1000, 1000\}, \{1000, 5000\}, \{5000, 1000\}$.