In this supplementary material, we present:

- **A:** Post-training weight distribution.
- **B:** Detailed derivation for gradient filtering described in Section 3.
- **C:** Detailed proof for Proposition 1 in Section 4.1.
- **D:** Visualized computation analysis for ResNet18.
- **E:** Detailed experimental setup for Section 5.1.
- **F:** More experimental results for Semantic Segmentation in Section 5.3.
- **G:** More experimental results for hyper-parameter exploration on CIFAR datasets in Section 5.4.
- **H:** Experimental results for combining gradient filtering (our method) with existing INT8 gradient quantization approaches [4, 7].
- **I:** More experimental results for on-device performance evaluation in Section 5.5.

A. Post-training Weight Distribution

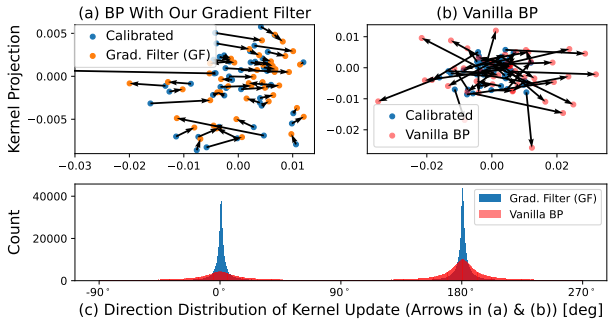


Figure 7. PCA projections of convolution kernels in the bottleneck layer of UperNet. Each point represents a 3×3 kernel. (a-b) compare the kernel before training (calibrated) and kernel trained with vanilla BP and our GF. (c) shows the distribution of directions [degree] in which the kernel was updated during training.

Figure 7 shows the PCA projections of convolution kernels in the bottleneck layer of an UperNet. Since our gradient filter (GF) only keeps the low-frequency part of the gradient signal (see Equation (7)), after applying the gradient filter, only the low-frequency part of the model is updated. As a result, as shown in Figure 7 (a) and (c), using the gradient filter limits the weights update to horizontal directions ($0°$ and $180°$), as opposed to using vanilla back propagation (BP) where all directions are involved (Figure 7 (b) and (c)).

B. Gradient Filtering Derivation

In this section, we present the complete derivations for Equation (3) and Equation (5) in Section 3, namely the back propagation with gradient filtering. For convenience, Table 6 (reproduced from Table 1 in paper) lists commonly used symbols.

B.1. Gradient Filtering

We have:

$$\tilde{g}_y[n, c_o, h, w] = \frac{1}{r^2} \sum_{h=\lfloor i/r \rfloor r}^{\lceil i/r \rceil r} \sum_{w=\lfloor j/r \rfloor r}^{\lceil j/r \rceil r} g_y[n, c_o, i, j] \quad (17)$$

Thus, for any entry in the approximated gradient $\tilde{g}_y$, the value equals to the average of all neighboring elements within the same $r \times r$ patch, as shown in Figure 2 in the main manuscript. For the approximated gradient $\tilde{g}_y$ with batch size n, channel c, resolution (H_y, W_y), there will be $(n \times c \times \lceil \frac{H_y}{r} \rceil \times \lceil \frac{W_y}{r} \rceil)$ unique numbers in $\tilde{g}_y$. To simplify the following derivations, we rewrite the approximated gradient $\tilde{g}_y$ as follows:

$$\tilde{g}_y^p[n, c_o, h_p, w_p, i, j] = \tilde{g}_y[n, c_o, h_p * r + i, w_p * r + j] \quad (18)$$

where (h_p, w_p) is the position of the patch and (i, j) is the offset within the patch. Since every element in the same patch has the exact same value, we denote this unique value with $\tilde{g}_y^u$, *i.e.*,

$$\tilde{g}_y^u[n, c_o, h_p, w_p] = \tilde{g}_y^p[n, c_o, h_p, w_p, i, j], \forall 0 \leq i, j < r \quad (19)$$

C_x	Number of channels of x
W_x, H_x	Width and height of x
θ	Convolution kernel
θ'	Rotated θ, *i.e.*, $\theta' = \text{rot180}(\theta)$
r	Patch size ($r \times r$)
g_x, g_y, g_θ	Gradients w.r.t. x, y, θ
$\tilde{g}_y$	Approximated gradient g_y
$\tilde{x}, \tilde{\theta}'$	Sum of x and θ' over spatial dimensions (height and width)
$x[n, c_i, h, w]$	Element for feature map x at batch n, channel c_i, pixel (h, w)
$\theta[c_o, c_i, u, v]$	Element for convolution kernel θ at output channel c_o, input channel c_i, position (u, v)

Table 6. Table of symbols we use.

B.2. Approximation for Rotated Convolution Kernel θ'

$$\begin{aligned}\tilde{\theta}'[c_o, c_i] &= \sum_{u,v} \theta'[c_o, c_i, u, v] \\ &= \sum_{u,v} \text{rot180}(\theta)[c_o, c_i, u, v] \\ &= \sum_{u,v} \theta[c_o, c_i, u, v]\end{aligned} \tag{20}$$

B.3. Approximation for Input Feature x

$$\tilde{x}[n, c_i, h, w] = \sum_{h=\lfloor i/r \rfloor r}^{\lceil i/r \rceil r} \sum_{w=\lfloor j/r \rfloor r}^{\lceil j/r \rceil r} x[n, c_i, i, j] \tag{21}$$

Thus for every entry in approximated feature map $\tilde{x}$, the value equal to the sum of all neighboring elements within the same $r \times r$ patch. Following the definition of the gradient filter in Section B.1, we use the following symbols to simplify the derivation:

$$\tilde{x}^p[n, c_i, h_p, w_p, i, j] = \tilde{x}[n, c_i, h_p * r + i, w_p * r + j] \tag{22}$$

and

$$\tilde{x}^u[n, c_i, h_p, w_p] = \tilde{x}^p[n, c_i, h_p, w_p, i, j], \forall 0 \leq i, j < r \tag{23}$$

B.4. Boundary Elements

As mentioned in Section 3, given the structure created by the gradient filters, the gradient propagation in a convolution layer can be simplified to weights summation and multiplication with few unique gradient values. This is true for *all elements* far away from the patch boundary because for these elements, the rotated kernel θ' only covers the elements from the same patch, which have the same value, thus the computation can be saved. However, for the elements close to the boundary, this is not true, since when convolving with boundary gradient elements, the kernel may cover multiple patches with multiple unique values instead of just one. To eliminate the extra computation introduced by the boundary elements, we pad each patch sufficiently such that every element is far away from boundary:

$$\tilde{g}_y^p[n, c_i, h_p, w_p, i, j] = \tilde{g}_y^u[n, c_i, h_p, w_p], \forall i, j \in \mathbb{Z} \tag{24}$$

For example, with the patch size 4×4, the element at the spatial position $(3, 3)$ is on the boundary, so when we calculate $\tilde{g}_x[n, c_i, 3, 3]$ by convolving the rotated kernel θ' with the approximated gradient $\tilde{g}_y$:

$$\tilde{g}_x[n, c_i, 3, 3] = \sum_{i,j} \theta'[c_o, c_i, i, j] \tilde{g}_y[n, c_o, 3+i, 3+j] \tag{25}$$

values of $\tilde{g}_y$ are from multiple patches and have different values (*e.g.*, $\tilde{g}_y[n, c_o, 3, 3]$ is from patch $(0, 0)$ while $\tilde{g}_y[n, c_o, 4, 4]$ is from patch $(1, 1)$; they have different values). In our method, we simplify the Equation (25) by rewriting it in the following way:

$$\begin{aligned}&\tilde{g}_x[n, c_i, 3, 3] \\ &\approx \sum_{i,j=-1}^{1} \theta'[c_o, c_i, i, j] \tilde{g}_y^p[n, c_o, \lfloor \frac{3}{4} \rfloor, \lfloor \frac{3}{4} \rfloor, 3+i, 3+j]\end{aligned} \tag{26}$$

$$= \sum_{i,j=-1}^{1} \theta'[c_o, c_i, i, j] \tilde{g}_y^u[n, c_o, \lfloor \frac{3}{4} \rfloor, \lfloor \frac{3}{4} \rfloor] \tag{27}$$

$$= \sum_{i,j=-1}^{1} \theta'[c_o, c_i, i, j] \tilde{g}_y^u[n, c_o, 0, 0] \tag{28}$$

where Equation (26) is derived from Equation (25) by considering that patch $(0, 0)$ is sufficiently padded so that for elements with all offsets $(3 + i, 3 + j)$, they have the same value, which is the unique value $g_y^u[n, c_o, 0, 0]$.

For approximated input feature map $\tilde{x}$, we apply the same approximation for the boundary elements.

B.5. Gradient w.r.t. Input (Equation (3) in Section 3.4)

$$\tilde{g}_x[n, c_i, h, w] \tag{29}$$

$$= \sum_{c_o,u,v} \theta[c_o, c_i, -u, -v] \tilde{g}_y[n, c_o, h + u, w + v] \tag{30}$$

$$\begin{aligned}&\approx \sum_{c_o,u,v} \theta[c_o, c_i, -u, -v] \cdot \\ &\quad \tilde{g}_y^p[n, c_o, \lfloor \frac{h}{r} \rfloor, \lfloor \frac{w}{r} \rfloor, (h \bmod r) + u, (w \bmod r) + v]\end{aligned} \tag{31}$$

$$= \sum_{c_o,u,v} \theta[c_o, c_i, -u, -v] \tilde{g}_y^u[n, c_o, \lfloor \frac{h}{r} \rfloor, \lfloor \frac{w}{r} \rfloor] \tag{32}$$

$$= \sum_{c_o} \tilde{g}_y^u[n, c_o, \lfloor \frac{h}{r} \rfloor, \lfloor \frac{w}{r} \rfloor] \sum_{u,v} \theta[c_o, c_i, -u, -v] \tag{33}$$

$$= \sum_{c_o} \tilde{g}_y^u[n, c_o, \lfloor \frac{h}{r} \rfloor, \lfloor \frac{w}{r} \rfloor] \tilde{\theta}'[c_o, c_i] \tag{34}$$

By expanding $\tilde{g}_y^u$ to $\tilde{g}_y$, we have:

$$\tilde{g}_x[n, c_i, h, w] = \sum_{c_o} \tilde{g}_y[n, c_o, h, w] \odot \tilde{\theta}'[c_o, c_i] \tag{35}$$

which is the Equation (3) in Section 3 in the paper.

From Equation (30) to Equation (32), we consider that the patch in the approximated gradient $\tilde{g}_y$ is padded sufficiently so they have the same value for all possible offsets

$((h \bmod r)+u, (w \bmod r)+v)$. If there is only one input channel and output channel for the convolutional layer as the Figure 2 in the paper shows, then Equation (34) become an element-wise multiplication, which is Equation (35) (also the Equation (3) in the Section 3.4).

B.6. Gradient w.r.t. Convolution Kernel (Equation (5) in the Section 3.4)

$$\tilde{g}_\theta[c_o, c_i, u, v] \tag{36}$$

$$= \sum_{n,h,w} x[n, c_i, h+u, w+v]\tilde{g}_y[n, c_o, h, w] \tag{37}$$

$$\approx \sum_{n,h,w} \tilde{x}^p[n, c_i, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor, (h \bmod r)+u, (w \bmod r)+v] \cdot \tilde{g}_y^u[n, c_o, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor] \tag{38}$$

$$= \sum_{n,h,w} \tilde{x}^u[n, c_i, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor]\tilde{g}_y^u[n, c_o, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor] \tag{39}$$

$$= \sum_{n,h,w} \tilde{x}^u[n, c_i, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor]\tilde{g}_y^u[n, c_o, \lfloor\frac{h}{r}\rfloor, \lfloor\frac{w}{r}\rfloor] \tag{40}$$

By expanding $\tilde{x}^u$ and $\tilde{g}_y^u$ to $\tilde{x}$ and $\tilde{g}_y$, respectively, we have:

$$\tilde{g}_\theta[c_o, c_i, u, v] = \sum_{n,i,j} \tilde{x}[n, c_i, i, j]\tilde{g}_y[n, c_o, i, j] \tag{41}$$

which is precisely Equation (5) in Section 3.

From Equation (37) to Equation (39), we consider that the patch in the approximated input feature map $\tilde{x}$ is padded sufficiently thus they have the same value for all possible offsets $((h \bmod r)+u, (w \bmod r)+v)$. For every given input/output channel pair (c_o, c_i), Equation (40) represents the Frobenius inner product between $\tilde{x}^u$ and $\tilde{g}_y^u$.

C. Detailed Proof for Proposition 1

In this section, we provide more details to the proof in Section 4.1. We use G_x, G_y and Θ to denote the gradients g_x, g_y and the convolution kernel θ in the *frequency domain*, respectively. $G_x[u, v]$ is the spectrum value at frequency (u, v) and δ is the 2D discrete Dirichlet function. Without losing generality and to simplify the proof, we consider the batch size is 1, the number of input/output channels is 1, namely $C_x = C_y = 1$, and there is only one patch in $\tilde{g}_y$.

The gradient returned by the gradient filtering can be written as:

$$\tilde{g}_y = \frac{1}{r^2} 1_{r\times r} \circledast g_y \tag{42}$$

where $\circledast$ denotes convolution. By applying the discrete Fourier transformation, Equation (42) can be rewritten in the frequency domain as:

$$\tilde{G}_y[u, v] = \frac{1}{r^2}\delta[u, v]G_y[u, v] \tag{43}$$

$\tilde{g}_y$ is the approximation for g_y(so the ground truth for $\tilde{g}_y$ is g_y), and the SNR of $\tilde{g}_y$ equals to:

$$\begin{aligned}\text{SNR}_{\tilde{g}_y} &= \left(\frac{\sum_{(u,v)}(G_y[u, v], -\tilde{G}_y[u, v])^2}{\sum_{(u,v)} G_y^2[u, v]}\right)^{-1} \\ &= \left(\frac{\sum_{(u,v)}(G_y[u, v] - \frac{1}{r^2}\delta[u, v]G_y[u, v])^2}{\sum_{(u,v)} G_y^2[u, v]}\right)^{-1}\end{aligned} \tag{44}$$

where the numerator can be written as:

$$\begin{aligned}&\sum_{(u,v)} (G_y[u, v] - \frac{1}{r^2}\delta[u, v]G_y[u, v])^2 \\ &= \sum_{(u,v)\neq(0,0)} (G_y[u, v] - \frac{1}{r^2}\delta[u, v]G_y[u, v])^2 \\ &\quad + (G_y[0, 0] - \frac{1}{r^2}\delta[0, 0]G_y[0, 0])^2\end{aligned} \tag{45}$$

Because $\delta[u, v] = \begin{cases} 1 & (u, v) = (0, 0) \\ 0 & (u, v) \neq (0, 0) \end{cases}$, Equation (45) can be written as:

$$\begin{aligned}&\sum_{(u,v)\neq(0,0)} G_y^2[u, v] + \frac{(r^2-1)^2}{r^4}G_y^2[0, 0] \\ &= \sum_{(u,v)\neq(0,0)} G_y^2[u, v] + G_y^2[0, 0] - G_y^2[0, 0] \\ &\quad + \frac{(r^2-1)^2}{r^4}G_y^2[0, 0] \\ &= \sum_{(u,v)} G_y^2[u, v] - \frac{2r^2-1}{r^4}G_y^2[0, 0]\end{aligned} \tag{46}$$

By substituting the numerator in Equation (44) with Equation (46), we have:

$$\begin{aligned}\text{SNR}_{\tilde{g}_y} &= \left(\frac{\sum_{(u,v)} G_y^2[u, v] - \frac{2r^2-1}{r^4}G_y^2[0, 0]}{\sum_{(u,v)} G_y^2[u, v]}\right)^{-1} \\ &= \left(1 - \frac{2r^2-1}{r^4}\frac{G_y^2[0, 0]}{\sum_{(u,v)} G_y^2[u, v]}\right)^{-1} \\ &= \left(1 - \frac{2r^2-1}{r^4}\frac{\text{Energy of DC Component in } G_y}{\text{Total Energy}^5 \text{ in } G_y}\right)^{-1}\end{aligned} \tag{47}$$

For the convolution layer, the gradient w.r.t. approximated variable $\tilde{x}$ in the frequency domain is:

$$\begin{aligned}\tilde{G}_x[u, v] &= \Theta[-u, -v]\tilde{G}_y[u, v] \\ &= \frac{1}{r^2}\Theta[-u, -v]\delta[u, v]G_y[u, v]\end{aligned} \tag{48}$$

[5] As reminder, the total energy of a signal is the sum of energy in DC component and energy in AC components.

and its ground truth is:

$$G_x[u,v] = \Theta[-u,-v]G_y[u,v] \tag{49}$$

Similar to Equation (47), the SNR of $g_{\tilde{x}}$ is:

$$\begin{aligned}\text{SNR}_{\tilde{g}_x} &= (1 - \frac{2r^2-1}{r^4}\frac{\Theta^2[0,0]G_y^2[0,0]}{\sum_{(u,v)}\Theta^2[u,v]G_y^2[u,v]})^{-1} \\ &= (1 - \frac{2r^2-1}{r^4}\frac{G_x^2[0,0]}{\sum_{(u,v)}G_x^2[u,v]})^{-1} \\ &= (1 - \frac{2r^2-1}{r^4}\frac{\text{Energy of DC Component in } G_x}{\text{Total Energy}^6 \text{in } G_x})^{-1}\end{aligned} \tag{50}$$

Equation (50) can be rewritten as:

$$\begin{aligned}\frac{r^4(1-\text{SNR}_{\tilde{g}_x}^{-1})}{2r^2-1} &= \frac{(\Theta[0,0]G_y[0,0])^2}{\sum_{(u,v)}(\Theta[-u,-v]G_y[u,v])^2} \\ &= \frac{G_y^2[0,0]}{\sum_{(u,v)}(\frac{\Theta[-u,-v]}{\Theta[0,0]}G_y[u,v])^2}\end{aligned} \tag{51}$$

Besides, the proposition's assumption (the DC component dominates the frequency spectrum of Θ) can be written as:

$$\frac{\Theta^2[0,0]}{\max_{(u,v)\neq(0,0)}\Theta^2[u,v]} \geq 1 \tag{52}$$

which is:

$$\forall(u,v), \frac{\Theta^2[-u,-v]}{\Theta^2[0,0]} \leq 1 \tag{53}$$

thus, by combining Equation (51) and Equation (53), we have:

$$\begin{aligned}\frac{r^4(1-\text{SNR}_{\tilde{g}_x}^{-1})}{2r^2-1} &= \frac{G_y^2[0,0]}{\sum_{(u,v)}(\frac{\Theta[-u,-v]}{\Theta[0,0]}G_y[u,v])^2} \\ &\geq \frac{G_y^2[0,0]}{\sum_{(u,v)}(G_y[u,v])^2} \\ &= \frac{r^4(1-\text{SNR}_{\tilde{g}_y}^{-1})}{2r^2-1}\end{aligned} \tag{54}$$

which means that: $\text{SNR}_{\tilde{g}_x} \geq \text{SNR}_{\tilde{g}_y}$. This completes our proof for error analysis.■

In conclusion, as the gradient propagates, the noise introduced by the gradient filter becomes weaker and weaker compared to the real gradient signal. This property ensures that the error in gradient has only a limited influence on the quality of BP.

This proof can be extended to the more general case where batch size and the number of channels are greater than 1 by introducing more dimensions (*i.e.*, batch dimension, channel dimension) into all equations listed above.

D. Computation Analysis for ResNet18

In this section, we provide one more example for computation analysis in Section 4.2. Figure 8 shows the computation required by the convolution layers from ResNet18 with different patch sizes for gradient filtering. With reduced unique elements, our approach reduces the number of computations to $1/r^2$ of standard BP method; with structured gradient, our approach further reduces the number of computations to about $1/(r^2 H_\theta W_\theta)$ of standard BP method.

E. Detailed Experimental Setup

In this section, we extend the experimental setup in Section 5.1.

E.1. ImageNet Classification

E.1.1 Environment

ImageNet related experiments are conducted on IBM Power System AC922, which is equipped with a 40-core IBM Power 9 CPU, 256 GB DRAM and 4 NVIDIA Tesla V100 16GB GPUs. We use PyTorch 1.9.0 compiled with CUDA 10.1 as the deep learning framework.

E.1.2 Dataset Split

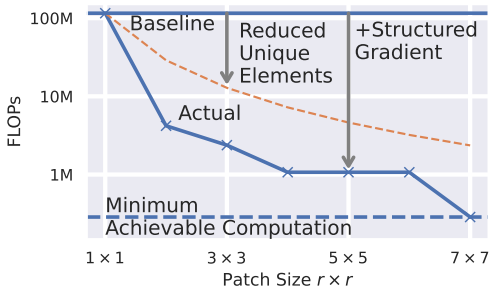
We split the dataset into two non-i.i.d. partitions following the FedAvg method [24]. The label distribution is shown in Figure 9. Among all 1000 classes for the ImageNet, pretrain and finetune partitions overlap on only 99 classes, which suggests that our method can efficiently adapt the CNN model to data collected from new environments. For each partition, we randomly select 80% data as training data and 20% as validation data.

E.1.3 Pretraining

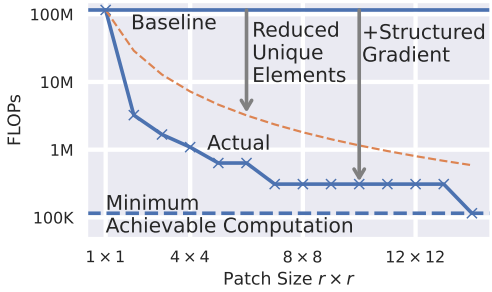
We pretrain ResNet 18, ResNet 34, MobileNet-V2 and MCUNet with the same configuration. We use SGD optimizer. The learning rate of the optimizer starts at 0.05 and decays according to cosine annealing method [22] during training. Additionally, weight decay is set to 1×10^{-4} and momentum is set to 0.9. We set batch size to 64. We randomly resize, randomly flip and normalize the image for data augmentation. We use cross entropy as loss function. Models are trained for 200 epochs and the model with the highest validation accuracy is kept for finetuning. Table 7 shows the pretrain accuracy.

E.1.4 Finetuning

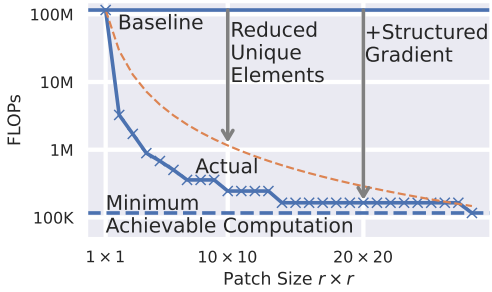
We adopt the hyper-parameter (*e.g.*, momentum, weight decay, etc.) from pretraining. Several changes are made: models are finetuned for 90 epochs instead of 200; we apply



(a) Last convolutional layer in block 4 of ResNet18 with 512 input/output channels; the resolution of input feature map is 7×7.



(b) Last convolutional layer in block 3 of ResNet18 with 256 input/output channels; the resolution of input feature map is 14×14.



(c) Last convolutional layer in block 2 of ResNet18 with 128 input/output channels; the resolution of input feature map is 28×28.

Figure 8. Computation analysis for three convolution layers in of ResNet18 model. Since convolutional layers in every block of ResNet18 is similar, we use the last convolutional layer as the representative of all convolutional layers in the block. Minimum achievable computation is presented in Equation (16) in the paper. By reducing the number of unique elements, computations required by our approach drop to about $1/r^2$ compared with the standard BP method. By combining it ("+" in the figure) with structured gradient map, computations required by our approach drop further.

Model	Accuracy	Model	Accuracy
ResNet-18	73.5%	MobileNet-V2	74.3%
ResNet-34	76.4%	MCUNet	71.4%

Table 7. Model pretraining accuracy on ImageNet.

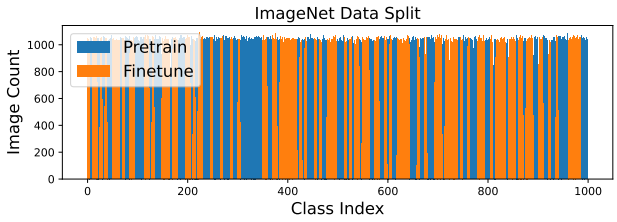


Figure 9. Label distribution for pretraining and finetuning datasets. Pretraining and finetuning partitions are split from ImageNet dataset.

L2 gradient clipping with threshold 2.0; linear learning rate warm-up for 4 epochs is introduced at the beginning of finetuning, *i.e.*, for the first 4 epochs, the learning rate grows linearly up to 0.05, then the learning rate decays according to cosine annealing method in the following epochs. Of note, to ensure a fair comparison, we use the same hyperparameters for all experiments, regardless of model type and training strategy.

E.2. CIFAR Classification

E.2.1 Environment

CIFAR related experiments are conducted on a GPU workstation with a 64-core AMD Ryzen Threadripper PRO 3995WX CPU, 512 GB DRAM and 4 NVIDIA RTX A6000 GPUs. We use PyTorch 1.12.0 compiled with CUDA 11.6 as the deep learning framework.

E.2.2 Dataset Split

We split the dataset into two non-i.i.d. partitions following FedAvg method. The label distribution is shown in Figure 10. For CIFAR10, pretrain and finetune partitions overlap on 2 classes out of 10 classes in total. For CIFAR100, pretrain and finetune partitions overlap on 6 classes out of 100 classes.

E.2.3 Pretraining

We pretrain ResNet18 and ResNet34 with the same configuration. We use the ADAM optimizer with a learning rate of 3×10^{-4} and weight decay 1×10^{-4} with no learning rate scheduling method. We use cross entropy as loss function. We set batch size to 128, and normalize the data before feeding it to the model. Models are trained for 30 and 50 epochs for CIFAR10 and CIFAR100, respectively. Then, the model with the highest accuracy is kept for finetuning. Table 8 shows the pretrain accuracy.

Figure 10. Label distribution for pretraining and finetuning datasets on CIFAR10 and CIFAR100. Pretraining and finetuning partitions are split from CIFAR10/100, respectively.

	ResNet18	ResNet34
CIFAR10	95.1%	97.6%
CIFAR100	75.5%	83.5%

Table 8. Model pretraining accuracy on CIFAR10/100.

E.2.4 Finetuning

We adopt the training configuration from PSQ [7] with some changes. We use cross entropy loss with SGD optimizer for training. The learning rate of the optimizer starts at 0.05 and decays according to cosine annealing method during training. Momentum is set to 0 and weight decay is set to 1×10^{-4}. We apply L2 gradient clipping with a threshold 2.0. Batch normalization layers are fused with convolution layers before training, which is a common technique for inference acceleration.

E.3. Semantic Segmentation

E.3.1 Environment

ImageNet related experiments are conducted on IBM Power System AC922, which is equipped with a 40-core IBM Power 9 CPU, 256 GB DRAM and 4 NVIDIA Tesla V100 16GB GPUs. We use PyTorch 1.9.0 compiled with CUDA 10.1 as the deep learning framework. We implement our method based on MMSegmentation 0.27.0.

E.3.2 Pretraining

We use models pretrained by MMSegmentation. Considering that the numbers of classes, image statistics, and model hyper-parameters may be different when applying on different datasets, we calibrate the model before finetuning. We use SGD optimizer. The learning rate of the optimizer starts at 0.01 and decays exponentially during training. Additionally, weight decay is set to 5×10^{-4} and momentum is set to 0.9. We set batch size to 8. We randomly crop, flip and photo-metric distort and normalize the image for data augmentation. We use cross entropy as loss function. For DeepLabV3, FCN, PSPNet and UPerNet, we calibrate the classifier (*i.e.*, the last layer) and statistics in batch normalization layers for 1000 steps on the finetuning dataset. For DeepLabV3-MobileNetV2 and PSPNet-MobileNetV2, because the number of channels for convolutional layers in the decoder are different for models applied on different datasets, we calibrate the decoder and statistics in batch normalization layers for 5000 steps on the finetuning dataset.

E.3.3 Finetuning

We finetune all models with the same configuration. We use the SGD optimizer. The learning rate of the optimizer starts at 0.01 and decays according to cosine anneling method during training. Additionally, weight decay is set to 5×10^{-4} and momentum is set to 0.9. We set batch size to 8. We randomly crop, flip and photo-metric distort and normalize the image for data augmentation. We use cross entropy as loss function. Models are finetuned for 20000 steps. Experiments are repeated three times with random seed 233, 234 and 235.

E.4. On-device Performance Evaluation

E.4.1 NVIDIA Jetson Nano

We use NVIDIA Jetson Nano with quad-core Cortex-A57, 4 GB DRAM, 128-core Maxwell edge GPU for performance evaluation on both edge CPU and edge GPU. We use the aarch64-OS Ubuntu 18.04.6 provided by NVIDIA. During evaluation, the frequencies for CPU and GPU are 1.5 GHz and 921 MHz, respectively. Our code and library MKLDNN (a.k.a. OneDNN) are compiled on Jetson Nano with GCC 7.5.0, while libraries CUDA and CUDNN are compiled by NVIDIA. For CPU evaluations, our code and baseline are implemented with MKLDNN v2.6. For GPU evaluations, our code and baseline are implemented with CUDA 10.2 and CUDNN 8.2.1.

Before the evaluation for every test case, we warm up the device by running the test once. Then we repeat the test 10 times and report the average value for latency, energy consumption, etc.

Energy consumption is obtained by reading the embedded power meter in Jetson Nano every 20 ms.

E.4.2 Raspberry Pi-3b

We use Raspberry Pi-3b with quad-core Cortex-A53, 1 GB DRAM for performance evaluation on CPU. We use

Pretrain: ADE20K Finetune: VOC12Aug													
UPerNet	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**	**PSPNet-M**	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**	**DLV3-M**	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**
Calibration	0	0	37.66	50.03	Calibration	0	0	30.93	52.01	Calibration	0	0	35.28	56.98
Vanilla BP	All	541.0	67.23[0.24]	79.79[0.45]	Vanilla BP	All	42.41	53.51[0.27]	67.01[0.19]	Vanilla BP	All	54.35	60.78[0.21]	74.10[0.40]
	5	503.9	72.01[0.09]	81.97[0.30]		5	12.22	48.88[0.11]	62.67[0.31]		5	14.77	51.51[0.09]	66.08[0.44]
	10	507.6	72.01[0.19]	81.83[0.44]		10	22.46	53.71[0.29]	67.93[0.32]		10	33.10	57.63[0.10]	71.93[0.41]
Ours	5	1.97	71.76[0.11]	81.57[0.07]	**Ours**	5	0.11	48.59[0.08]	62.28[0.30]	**Ours**	5	0.26	49.40[0.00]	64.13[0.54]
	10	2.22	71.78[0.23]	81.55[0.38]		10	0.76	52.77[0.37]	66.82[0.47]		10	1.40	55.14[0.15]	69.48[0.26]
Pretrain: ADE20K Finetune: Cityscapes														
UPerNet	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**	**PSPNet-M**	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**	**DLV3-M**	**#Layers**	**GFLOPs**	**mIoU**	**mAcc**
Calibration	0	0	34.15	42.45	Calibration	0	0	28.83	34.85	Calibration	0	0	41.33	48.65
Vanilla BP	All	1082.1	73.02[0.14]	81.01[0.20]	Vanilla BP	All	84.82	60.21[0.40]	67.72[0.68]	Vanilla BP	All	108.7	71.12[0.14]	79.81[0.04]
	5	1007.7	62.46[0.19]	72.62[0.27]		5	24.43	42.09[0.43]	48.70[0.49]		5	29.5	51.00[0.05]	59.20[0.03]
	10	1015.3	64.01[0.21]	73.11[0.32]		10	44.90	54.03[0.24]	61.48[0.10]		10	66.2	61.02[0.14]	69.80[0.06]
Ours	5	3.94	60.58[0.25]	70.67[0.32]	**Ours**	5	0.22	41.59[0.38]	48.10[0.41]	**Ours**	5	0.50	48.83[0.07]	56.87[0.08]
	10	4.43	62.14[0.24]	71.41[0.27]		10	1.51	49.10[0.49]	56.93[1.43]		10	2.74	50.22[1.01]	59.99[0.31]

Table 9. Experimental results for semantic segmentation task for UPerNet, DeepLabV3-MobileNetV2 (DLV3-M) and PSPNet-MobileNetV2 (PSPNet-M). Models are pretrained on ADE20K dataset and finetuned on augmentated Pascal VOC12 dataset and Cityscapes dataset respectively. "#Layers" is short for "the number of *active* convolutional layers" that are trained. Strategy "Calibration" shows the accuracy when only the classifier and normalization statistics are updated to adapt differences (*e.g.* different number of classes) between pretraining dataset and finetuning dataset.

No.	#Input Channel	#Output Channel	Input Width	Input Height
0	128	128	120	160
1	256	256	60	80
2	512	512	30	40
3	512	512	14	14
4	256	256	14	14
5	128	128	28	28
6	64	64	56	56

Table 10. Layer configuration for test cases in Figure 6 in Section 5.5 in the paper.

the aarch64-OS Raspberry Pi OS. During evaluation, the frequency for CPU is 1.2 GHz. Our code and library MKLDNN are compiled on Raspberry Pi with GCC 10.2. Our code and baseline are implemented with MKLDNN v2.6.

Before the evaluation for every test case, we warm up the device by running the test once. Then we repeat the test 10 times and report the average value for latency, etc.

E.4.3 Desktop

We use a desktop PC with Intel 11900KF CPU, 32 GB DRAM and RTX 3090 Ti GPU for perforamce evaluation on both desktop CPU and desktop GPU. We use x86_64-OS Ubuntu 20.04. During evaluation, the frequencies for CPU and GPU are 4.7 GHz and 2.0 GHz respectively. Our code is compiled with GCC 9.4.0. MKLDNN is compiled by Anaconda (tag omp_h13be974_0). CUDA and CUDNN are compiled by NVIDIA. For CPU evaluations, our code and baseline are implemented with MKLDNN v2.6. For GPU evaluations, our code and baseline are implemented with CUDA 11.7 and CUDNN 8.2.1.

Before the evaluation for every test case, we warm up the device by running the 10 times. Then we repeat the test 200 times and report the average value for latency, etc.

E.4.4 Test Case Configurations

Table 10 lists the configurations for test cases shown in Figure 6 in the paper. In addition to the parameters shown in the table, for all test cases, we set the batch size to 32, kernel size to 3×3, padding and stride to 1.

F. More Results for Semantic Segmentation

In this section, we extend the experimental results shown in Section 5.3 (Table 3). Table 9 shows the experimental results for UPerNet, PSPNet-MobileNetV2 (PSPNet-M) and DeepLabV3-MobileNetV2 (DLV3-M) on two pairs of pretraing and finetuning datasets. These results further show the effectiveness of our method on a dense prediction task.

G. More Results for CIFAR10/100 with Different Hyper-Parameter Selections

In this section, we extend the experimental results shown in Section 5.4 (Figure 4). Table 11 shows the experimental results for ResNet18 and ResNet34 on CIFAR datasets. For every model, we test our method with different patch sizes for gradient filtering and different numbers of *active* convolutional layers (#Layers in Table 11, *e.g.*, if #Layers equals to 2, the last two convolutional layers are trained while other layers are frozen). These results further support the qualitative findings in Section 5.4.

H. Results for Combining Gradient Filtering with Gradient Quantization

In this section, we provide experimental results for combining our method, *i.e.* gradient filtering, with gradient quantization. Table 12 shows experimental results for ResNet18 and ResNet32 with gradient quantization methods PTQ [4] and PSQ [7] and different hyper-parameters.

CIFAR10								CIFAR100							
ResNet18	#Layers	ACC[%]	FLOPs	**ResNet34**	#Layers	ACC[%]	FLOPs	**ResNet18**	#Layers	ACC[%]	FLOPs	**ResNet34**	#Layers	ACC[%]	FLOPs
Vanilla BP	1	91.7	128.25M	Vanilla BP	1	94.2	128.25M	Vanilla BP	1	73.8	128.39M	Vanilla BP	1	76.9	128.39M
	2	93.6	487.68M		2	96.6	487.68M		2	77.6	487.82M		2	82.0	487.82M
	3	93.7	847.15M		3	96.6	847.13M		3	77.6	847.29M		3	82.1	847.27M
	4	94.4	1.14G		4	96.8	1.21G		4	78.0	1.14G		4	83.0	1.21G
+Gradient Filter R2	1	91.5	8.18M	+Gradient Filter R2	1	94.2	8.18M	+Gradient Filter R2	1	73.7	8.31M	+Gradient Filter R2	1	77.0	8.31M
	2	92.7	26.80M		2	96.6	26.80M		2	75.6	26.94M		2	81.1	26.94M
	3	92.8	45.45M		3	96.5	45.44M		3	75.6	45.59M		3	81.1	45.58M
	4	93.9	60.01M		4	96.6	64.07M		4	76.4	60.15M		4	82.0	64.21M
+Gradient Filter R4	1	91.4	1.88M	+Gradient Filter R4	1	94.3	1.88M	+Gradient Filter R4	1	73.7	2.02M	+Gradient Filter R4	1	76.9	2.02M
	2	92.7	7.93M		2	96.4	7.93M		2	74.9	8.07M		2	80.4	8.07M
	3	92.8	13.99M		3	96.4	13.98M		3	74.9	14.12M		3	80.4	14.12M
	4	93.3	19.12M		4	96.1	20.04M		4	75.2	19.26M		4	80.5	20.17M
+Gradient Filter R7	1	91.5	303.10K	+Gradient Filter R7	1	94.2	303.10K	+Gradient Filter R7	1	73.7	441.34K	+Gradient Filter R7	1	76.9	441.34K
	2	91.5	3.21M		2	95.8	3.21M		2	74.1	3.35M		2	80.4	3.35M
	3	91.7	6.12M		3	96.0	6.12M		3	74.1	6.26M		3	80.3	6.26M
	4	92.6	8.90M		4	96.0	9.03M		4	75.4	9.04M		4	80.3	9.17M

Table 11. Experimental results on CIFAR10 and CIFAR100 datasets for ResNet18 and ResNet34 with different hyper-parameter selections. "ACC" is short for accuracy. "#Layers" is short for "the number of *active* convolution layers". For example. #Layers equals to 2 means that only the last two convolutional layers are trained. "Gradient Filter R2/4/7" use proposed gradient filtering method with patch size 2×2, 4×4 and 7×7, respectively.

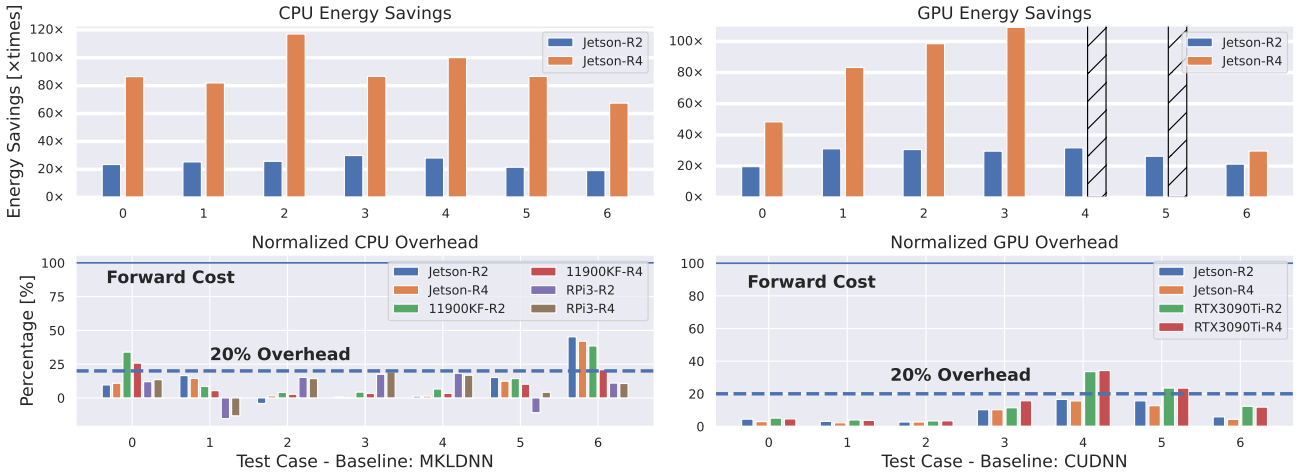


Figure 11. Energy savings and overhead resuls on multiple CPUs and GPUs under different test cases (*i.e.*, different input sizes, number of channels, etc..). For test case 4 and 5 with patch size 4×4 (Jetson-R4) on GPU, the latency of our method is too small to be captured by the power meter with a 20 ms sample rate so the energy savings data is not available. For most test cases with patch size 4×4, our method achieves over $80\times$ energy savings with less than 20% overhead.

Both forward propagation and backward propagation are quantized to INT8. These results support the wide applicability of our method.

I. More Results for On-device Performance Evaluation

In this section, we extend the experimental results shown in Section 5.5. Figure 11 shows the energy savings and overhead of our method. For most test cases with patch 4×4, we achieve over $80\times$ energy savings with less than 20% overhead on both CPU and GPU. Moreover, for the test case 1 on Raspberry Pi-3b CPU, the forward propagation is even faster when applied our method (which results in negtive overheads). These results further show that our method is practical for the real deployment of both high-performance and IoT applications.

CIFAR10								CIFAR100							
ResNet18				ResNet34				ResNet18				ResNet34			
Strategy	#Layers	ACC[%]	#OPs	Strategy	#Layers	ACC[%]	#OPs	Strategy	#Layers	ACC[%]	#OPs	Strategy	#Layers	ACC[%]	#OPs
PTQ	1	91.6	128.25M	PTQ	1	93.6	128.25M	PTQ	1	74.0	128.39M	PTQ	1	76.4	128.39M
	2	93.2	487.68M		2	96.2	487.68M		2	77.8	487.82M		2	80.3	487.82M
	3	93.5	847.15M		3	96.2	847.13M		3	77.9	847.29M		3	80.5	847.27M
	4	94.4	1.14G		4	96.5	1.21G		4	77.9	1.14G		4	82.2	1.21G
PTQ	1	91.4	8.18M	PTQ	1	93.5	8.18M	PTQ	1	73.9	8.31M	PTQ	1	76.5	8.31M
+Gradient	2	92.6	26.80M	+Gradient	2	95.9	26.80M	+Gradient	2	75.7	26.94M	+Gradient	2	80.0	26.94M
Filter	3	92.7	45.45M	Filter	3	96.0	45.44M	Filter	3	75.9	45.59M	Filter	3	80.1	45.58M
R2	4	93.7	60.01M	R2	4	96.2	64.07M	R2	4	76.3	60.15M	R2	4	80.9	64.21M
PTQ	1	91.3	1.88M	PTQ	1	93.6	1.88M	PTQ	1	73.7	2.02M	PTQ	1	76.5	2.02M
+Gradient	2	92.5	7.93M	+Gradient	2	95.6	7.93M	+Gradient	2	75.1	8.07M	+Gradient	2	79.5	8.07M
Filter	3	92.7	13.99M	Filter	3	95.6	13.98M	Filter	3	75.4	14.12M	Filter	3	79.5	14.12M
R4	4	93.4	19.12M	R4	4	95.6	20.04M	R4	4	76.1	19.26M	R4	4	80.5	20.17M
PTQ	1	91.2	303.10K	PTQ	1	93.6	303.10K	PTQ	1	73.7	441.34K	PTQ	1	76.5	441.34K
+Gradient	2	91.5	3.21M	+Gradient	2	95.5	3.21M	+Gradient	2	74.5	3.35M	+Gradient	2	79.4	3.35M
Filter	3	91.6	6.12M	Filter	3	95.4	6.12M	Filter	3	74.5	6.26M	Filter	3	79.5	6.26M
R7	4	92.6	8.90M	R7	4	95.5	9.03M	R7	4	75.3	9.04M	R7	4	79.6	9.17M
PSQ	1	91.4	128.25M	PSQ	1	93.6	128.25M	PSQ	1	73.9	128.39M	PSQ	1	76.4	128.39M
	2	93.3	487.68M		2	96.1	487.68M		2	77.7	487.82M		2	80.3	487.82M
	3	93.4	847.15M		3	96.2	847.13M		3	77.9	847.29M		3	80.5	847.27M
	4	94.5	1.14G		4	96.4	1.21G		4	78.0	1.14G		4	82.2	1.21G
PSQ	1	91.3	8.18M	PSQ	1	93.5	8.18M	PSQ	1	73.8	8.31M	PSQ	1	76.4	8.31M
+Gradient	2	92.6	26.80M	+Gradient	2	96.0	26.80M	+Gradient	2	76.0	26.94M	+Gradient	2	80.1	26.94M
Filter	3	92.8	45.45M	Filter	3	96.1	45.44M	Filter	3	75.9	45.59M	Filter	3	80.0	45.58M
R2	4	93.7	60.01M	R2	4	96.1	64.07M	R2	4	76.3	60.15M	R2	4	80.9	64.21M
PSQ	1	91.4	1.88M	PSQ	1	93.6	1.88M	PSQ	1	73.5	2.02M	PSQ	1	76.5	2.02M
+Gradient	2	92.6	7.93M	+Gradient	2	95.6	7.93M	+Gradient	2	75.3	8.07M	+Gradient	2	79.5	8.07M
Filter	3	92.7	13.99M	Filter	3	95.6	13.98M	Filter	3	75.1	14.12M	Filter	3	79.6	14.12M
R4	4	93.2	19.12M	R4	4	95.5	20.04M	R4	4	76.2	19.26M	R4	4	80.2	20.17M
PSQ	1	91.2	303.10K	PSQ	1	93.6	303.10K	PSQ	1	73.5	441.34K	PSQ	1	76.5	441.34K
+Gradient	2	91.4	3.21M	+Gradient	2	95.5	3.21M	+Gradient	2	74.4	3.35M	+Gradient	2	79.5	3.35M
Filter	3	91.6	6.12M	Filter	3	95.4	6.12M	Filter	3	74.5	6.26M	Filter	3	79.6	6.26M
R7	4	92.7	8.90M	R7	4	95.5	9.03M	R7	4	75.5	9.04M	R7	4	79.6	9.17M

Table 12. Experimental results for ResNet18 and ResNet34 with different gradient quantization methods (*i.e.*, PTQ [4] and PSQ [7]) and hyper-parameter selections on CIFAR10/100. Feature map, activation, weight and gradient are quantized to INT8. "ACC" is short for accuracy. "#Layers" is short for "the number of *active* convolution layers". For example. #Layers equals to 2 means that the last two convolutional layers are trained. "Gradient Filter R2/4/7" use proposed gradient filtering method with patch size 2×2, 4×4 and 7×7, respectively.