Appendix

A. ALQ Initialization

A.1. Initialization Algorithm

We adapt the network sketching in [9], and propose a structured sketching algorithm below for ALQ initialization (see Alg. 1)[2]. Here, the subscript of the layer index l is reintroduced for a layerwise elaboration in the pseudocode. This algorithm partitions the pretrained full precision weights $\boldsymbol{w}_l$ of the lth layer into G_l groups with the structures mentioned in A.2. The vectorized weights $\boldsymbol{w}_{l,g}$ of each group are quantized with $I_{l,g}$ linear independent binary bases (*i.e.* column vectors in $\boldsymbol{B}_{l,g}$) and corresponding coordinates $\boldsymbol{\alpha}_{l,g}$ to minimize the reconstruction error. This algorithm initializes the matrix of binary bases $\boldsymbol{B}_{l,g}$, the vector of floating-point coordinates $\boldsymbol{\alpha}_{l,g}$, and the scalar of integer bitwidth $I_{l,g}$ in each group across layers. The initial reconstruction error is upper bounded by a threshold σ. In addition, a maximum bitwidth of each group is defined as $\mathrm{I}_{\max}$. Both of these two parameters determine the initial bitwidth $I_{l,g}$.

Algorithm 1: Structured Sketching of Weights

```
Input: {w_l}_{l=1}^L, {G_l}_{l=1}^L, I_max, σ
Output: {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^L
for l ← 1 to L do
    for g ← 1 to G_l do
        Fetch and vectorize w_{l,g} from w_l ;
        Initialize ε = w_{l,g}, i = 0 ;
        B_{l,g} = [ ] ;
        while ||ε ⊘ w_{l,g}||_2^2 > σ and i < I_max do
            i = i + 1;
            β_i = sign(ε);
            B_{l,g} = [B_{l,g}, β_i];
            /* Find the optimal point
               spanned by B_{l,g}          */
            α_{l,g} = (B_{l,g}^T B_{l,g})^{-1} B_{l,g}^T w_{l,g} ;
            /* Update the residual
               reconstruction error        */
            ε = w_{l,g} − B_{l,g} α_{l,g} ;
        I_{l,g} = i;
```

Theorem A.1. The column vectors in $\boldsymbol{B}_{l,g}$ are linear independent.

Proof. The instruction $\boldsymbol{\alpha}_{l,g} = (\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{B}_{l,g})^{-1}\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{w}_{l,g}$ ensures $\boldsymbol{\alpha}_{l,g}$ is the optimal point in $\mathrm{span}(\boldsymbol{B}_{l,g})$ regarding the least square reconstruction error $\boldsymbol{\epsilon}$. Thus, $\boldsymbol{\epsilon}$ is orthogonal to $\mathrm{span}(\boldsymbol{B}_{l,g})$. The new basis is computed from the next iteration by $\boldsymbol{\beta}_i = \mathrm{sign}(\boldsymbol{\epsilon})$. Since $\mathrm{sign}(\boldsymbol{\epsilon}) \bullet \boldsymbol{\epsilon} > 0, \forall \boldsymbol{\epsilon} \neq \mathbf{0}$, we have $\boldsymbol{\beta}_i \notin \mathrm{span}(\boldsymbol{B}_{l,g})$. Thus, the iteratively generated column vectors in $\boldsymbol{B}_{l,g}$ are linear independent. This also means the square matrix of $\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{B}_{l,g}$ is invertible. □

[2] Circled operation in Alg. 1 means elementwise operations.

A.2. Experiments on Group Size

Researchers propose different structured quantization in order to exploit the redundancy and the tolerance in the different structures. Certainly, the weights in one layer can be arbitrarily selected to gather a group. Considering the extra indexing cost, in general, the weights are sliced along the tensor dimensions and uniformly grouped.

According to [9], the squared reconstruction error of a single group decays with Eq.(27), where $\lambda \geq 0$.

$$\|\boldsymbol{\epsilon}\|_2^2 \leq \|\boldsymbol{w}_g\|_2^2 (1 - \frac{1}{n-\lambda})^{I_g} \tag{27}$$

If full precision values are stored in floating-point datatype, *i.e.* 32-bit, the storage compression rate in one layer can be written as,

$$r_s = \frac{N \times 32}{I \times N + I \times 32 \times \frac{N}{n}} \tag{28}$$

where N is the total number of weights in one layer; n is the number of weights in each group, *i.e.* $n = N/G$; I is the average bitwidth, $I = \frac{1}{G}\sum_{g=1}^{G} I_g$.

We analyse the trade-off between the reconstruction error and the storage compression rate of different group size n. We choose the pretrained AlexNet [20] and VGG-16 [40], and plot the curves of the average (per weight) reconstruction error related to the storage compression rate of each layer under different sliced structures. We also randomly shuffle the weights in each layer, then partition them into groups with different sizes. We select one example plot which comes from the last convolution layer ($256 \times 256 \times 3 \times 3$) of AlexNet [20] (see Fig. 3). The pretrained full precision weights are provided by Pytorch [30].

We have found that there is not a significant difference between random groups and sliced groups (along original tensor dimensions). Only the group size influences the trade-off. We argue the reason is that one layer always contains thousands of groups, such that the points presented by these groups are roughly scattered in the n-dim space. Furthermore, regarding the deployment on a 32-bit general microprocessor, the group size should be larger than 32 for efficient computation. In short, a group size from 32 to 512 achieves relatively good trade-off between the reconstruction error and the storage compression.

These above demonstrated three structures in Fig. 3 do not involve the cross convolutional filters' computation, which leads to less run-time memory than other structures. Accordingly, for a convolution layer, grouping in channel-wise ($\boldsymbol{w}_{c,:,:,:}$), kernel-wise ($\boldsymbol{w}_{c,d,:,:}$), and pixel-wise ($\boldsymbol{w}_{c,:,h,w}$)

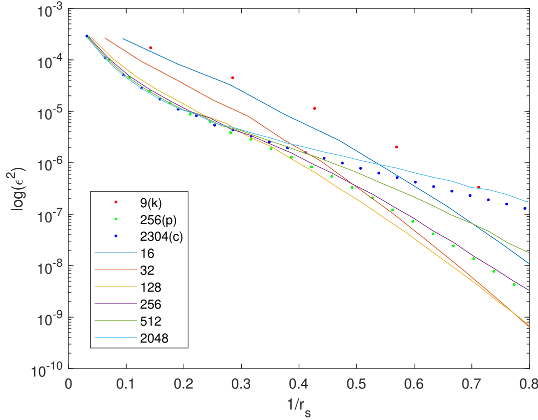

Figure 3. The curves about the logarithmic L2-norm of the average reconstruction error $\log(\|\boldsymbol{\epsilon}\|_2^2)$ related to the reciprocal of the storage compression rate $1/r_s$ (from the last convolution layer of AlexNet). The legend demonstrates the corresponding group sizes. 'k' stands for kernel-wise; 'p' stands for pixel-wise; 'c' stands for channel-wise.

are appropriate. Channel-wise $\boldsymbol{w}_{c,:}$ and subchannel-wise $\boldsymbol{w}_{c,d:d+n}$ grouping are suited for a fully connected layer. The most frequently used structures for current popular network are pixel-wise (convolution layers) and (sub)channel-wise (fully connected layers), which exactly coincide the bit-packing approach in [31], and could result in a more efficient deployment. Since many network architectures choose an integer multiple of 32 as the number of output channels in each layer, pixel-wise and (sub)channel-wise are also efficient for the current storage format in 32-bit microprocessors, *i.e*. in 4 Bytes (32-bit integer).

B. Pseudocode and Complexity Analysis

B.1. Pruning in α Domain

In each execution of Step 1 (Sec. 3.2), 30% of α_i's are pruned. Iterative pruning is realized in mini-batch (general 1 epoch in total). Due to the high complexity of sorting all $f_{\boldsymbol{\alpha},i}$, sorting is firstly executed in each layer, and the top-k% $f_{\boldsymbol{\alpha}_l,i}$ of the lth layer are selected to resort again for pruning. k is generally small, *e.g*. 1 or 0.5, which ensures that the pruned α_i's in one iteration do not come from a single layer. Again, $\boldsymbol{\alpha}_l$ is vectorized $\{\boldsymbol{\alpha}_{l,g}\}_{g=1}^{G_l}$; $\boldsymbol{B}_l$ is concatenated $\{\boldsymbol{B}_{l,g}\}_{g=1}^{G_l}$ in the lth layer. There are n_l weights in each group, and G_l groups in the lth layer.

The number of total layers is usually smaller than 100, thus, the sorting complexity mainly depends on the sorting in the layer, which has the largest $\text{card}(\boldsymbol{\alpha}_l)$. The number of the sorted element $f_{\boldsymbol{\alpha}_l,i}$, *i.e*. $\text{card}(\boldsymbol{\alpha}_l)$, is usually smaller than an order of 10^4 for a general network in ALQ.

The pruning step in Sec. 3.2 is demonstrated in Alg. 2. Here, assume that there are altogether T times pruning iterations in each execution of Step 1; the total number of α_i's across all layers is M_0 before pruning, *i.e*.

$$M_0 = \sum_l \sum_g \text{card}(\boldsymbol{\alpha}_{l,g}) \tag{29}$$

and the desired total number of α_i's after pruning is M_T.

Algorithm 2: Pruning in α Domain

Input: T, M_T, k, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, Training Data
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$
Compute M_0 **with** Eq.(29) ;
Determine the pruning number at each iteration $M_p = \text{round}(\frac{M_0 - M_T}{T})$;
for $t \leftarrow 1$ **to** T **do**
  for $l \leftarrow 1$ **to** L **do**
    Update $\hat{\boldsymbol{w}}_{l,g}^t = \boldsymbol{B}_{l,g}^t \boldsymbol{\alpha}_{l,g}^t$;
    Forward propagate convolution ;
  Compute the loss ℓ^t ;
  for $l \leftarrow L$ **to** 1 **do**
    Backward propagate gradient $\frac{\partial \ell^t}{\partial \hat{\boldsymbol{w}}_{l,g}^t}$;
    Compute $\frac{\partial \ell^t}{\partial \boldsymbol{\alpha}_{l,g}^t}$ **with** Eq.(14) ;
    Update momentums of AMSGrad in $\boldsymbol{\alpha}$ **domain** ;
    for $\alpha_{l,i}^t$ **in** $\boldsymbol{\alpha}_l^t$ **do**
      Compute $f_{\boldsymbol{\alpha}_l,i}^t$ **with** Eq.(13) ;
    Sort and select Top-k% $f_{\boldsymbol{\alpha}_l,i}^t$ **in ascending order** ;
  Resort the selected $\{f_{\boldsymbol{\alpha}_l,i}^t\}_{l=1}^{L}$ **in ascending order** ;
  Remove Top-M_p $\alpha_{l,i}^t$ **and their binary bases** ;
  Update $\{\{\boldsymbol{\alpha}_{l,g}^{t+1}, \boldsymbol{B}_{l,g}^{t+1}, I_{l,g}^{t+1}\}_{g=1}^{G_l}\}_{l=1}^{L}$;

B.2. Optimizing Binary Bases and Coordinates

Step 2 is also executed in batch training. In Step 2 (Sec. 3.3), 10^{-3} is used as the learning rate in optimizing $\boldsymbol{B}_g$, and gradually decays in each epoch; the learning rate is set to 10^{-5} in optimizing $\boldsymbol{\alpha}_g$, and also gradually decays in each epoch.

B.2.1 Optimizing $\boldsymbol{B}_g$ with Speedup

The extra complexity related to the original AMSGrad mainly comes from two parts, Eq.(19) and Eq.(23). Eq.(19) is also the most resource-hungry step of the whole pipeline, since it requires an exhaustive search. For each group, Eq.(19) takes both time and storage complexities of $O(n2^{I_g})$, and in general $n >> I_g \geq 1$. Since $\boldsymbol{H}^q$ is a diagonal

matrix, most of matrix-matrix multiplication in Eq.(23) is avoided through matrix-vector multiplication and matrix-diagonalmatrix multiplication. Thus, the time complexity trims down to $O(nI_g+nI_g^2+I_g^3+nI_g+n+n+nI_g+I_g^2) \doteq O(n(I_g^2+3I_g+2))$. In our settings, optimizing $\boldsymbol{B}_g$ with speedup usually takes around twice as long as optimizing $\boldsymbol{\alpha}_g$ (*i.e.* the original AMSGrad step).

Optimizing $\boldsymbol{B}_g$ with speedup (Sec. 3.3) is presented in Alg. 3. Assume that there are altogether Q iterations. It is worth noting that the bitwidth $I_{l,g}$ does not change in this step; only the binary bases $\boldsymbol{B}_{l,g}$ and the coordinates $\boldsymbol{\alpha}_{l,g}$ are updated over Q iterations.

Algorithm 3: Optimizing $\boldsymbol{B}_g$ with Speedup

```
Input: Q, {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^{L}, Training Data
Output: {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^{L}
for q ← 1 to Q do
    for l ← 1 to L do
        Update ŵ_{l,g}^q = B_{l,g}^q α_{l,g}^q ;
        Forward propagate convolution ;
    Compute the loss ℓ^q ;
    for l ← L to 1 do
        Backward propagate gradient ∂ℓ^q/∂ŵ_{l,g}^q ;
        Update momentums of AMSGrad ;
        for g ← 1 to G_l do
            Compute all values of Eq.(20) ;
            for j ← 1 to n_l do
                Update B_{l,g,j}^{q+1} according to the
                  nearest value (see Eq.(19)) ;
            Update α_{l,g}^{q+1} with Eq.(23) ;
```

B.2.2 Optimizing $\boldsymbol{\alpha}_g$

Since $\boldsymbol{\alpha}_g$ is floating-point value, the complexity of optimizing $\boldsymbol{\alpha}_g$ is the same as the conventional optimization step, (see Alg. 4). Assume that there are altogether P iterations. It is worth noting that both the bitwidth $I_{l,g}$ and the binary bases $\boldsymbol{B}_{l,g}$ do not change in this step; only the coordinates $\boldsymbol{\alpha}_{l,g}$ are updated over P iterations.

B.3. Whole Pipeline of ALQ

The whole pipeline of ALQ is demonstrated in Alg. 5.

For the initialization, the pretrained full precision weights (model) $\{\boldsymbol{w}_l\}_{l=1}^{L}$ is required. Then, we need to specify the structure used in each layer, *i.e.* the manner of grouping (for short marked with $\{G_l\}_{l=1}^{L}$). In addition, a maximum bitwidth $\mathrm{I}_{\max}$ and a threshold σ for the residual reconstruction error also need to be determined (see more details in A).

Algorithm 4: Optimizing $\boldsymbol{\alpha}_g$

```
Input: P, {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^{L}, Training Data
Output: {{α_{l,g}, B_{l,g}, I_{l,g}}_{g=1}^{G_l}}_{l=1}^{L}
for p ← 1 to P do
    for l ← 1 to L do
        Update ŵ_{l,g}^p = B_{l,g} α_{l,g}^p ;
        Forward propagate convolution ;
    Compute the loss ℓ^p ;
    for l ← L to 1 do
        Backward propagate gradient ∂ℓ^p/∂ŵ_{l,g}^p ;
        Compute ∂ℓ^p/∂α_{l,g}^p with Eq.(14) ;
        Update momentums of AMSGrad in α
          domain ;
        for g ← 1 to G_l do
            Update α_{l,g}^{p+1} with Eq.(21) ;
```

After initialization, we might need to retrain the model with several epochs of B.2 to recover the accuracy degradation caused by the initialization.

Then, we need to determine the number of outer iterations R, *i.e.* how many times the pruning step (Step 1) is executed. A pruning schedule $\{M^r\}_{r=1}^{R}$ is also required. M^r determines the total number of remained α_i's (across all layers) after the rth pruning step, which is also taken as the input M_T in Alg. 2. For example, we can build this schedule by pruning 30% of α_i's during each execution of Step 1, as,

$$M^{r+1} = M^r \times (1 - 0.3) \tag{30}$$

with $r \in \{0, 1, 2, \ ..., \ R-1\}$. M^0 represents the total number of α_i's (across all layers) after initialization.

For Step 1 (Pruning in $\boldsymbol{\alpha}$ Domain), other individual inputs include the total number of iterations T, and the selected percentages k for sorting (see Alg. 2). For Step 2 (Optimizing Binary Bases and Coordinates), the individual inputs includes the total number of iterations Q in optimizing $\boldsymbol{B}_g$ (see Alg. 3), and the total number of iterations P in optimizing $\boldsymbol{\alpha}_g$ (see Alg. 4).

B.4. STE with Loss-aware

In this section, we provide the details of the proposed *STE with loss-aware* optimizer. The training scheme of *STE with loss-aware* is similar as Optimizing $\boldsymbol{B}_g$ with Speedup (see B.2.1), except that it maintains the full precision weights $\boldsymbol{w}_g$. See the pseudocode of *STE with loss-aware* in Alg. 6.

For the layer l, the quantized weights $\hat{\boldsymbol{w}}_g$ is used during forward propagation. During backward propagation, the loss gradients to the full precision weights $\frac{\partial \ell}{\partial \boldsymbol{w}_g}$ are directly

Algorithm 5: Adaptive Loss-aware Quantization

Input: Pretrained FP Weights $\{\boldsymbol{w}_l\}_{l=1}^{L}$, Structures $\{G_l\}_{l=1}^{L}$, $\mathrm{I}_{\max}$, σ, T, Pruning Schedule $\{M^r\}_{r=1}^{R}$, k, P, Q, R, Training Data
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$

```
/* Initialization: */
```
Initialize $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$ **with Alg. 1** ;
for $r \leftarrow 1$ **to** R **do**
```
    /* Step 1: */
```
  Assign M^r **to the input** M_T **of Alg. 2** ;
  Prune in $\boldsymbol{\alpha}$ **domain with Alg. 2** ;
```
    /* Step 2: */
```
  Optimize binary bases with Alg. 3 ;
  Optimize coordinates with Alg. 4 ;

approximated with $\frac{\partial \ell}{\partial \hat{\boldsymbol{w}}_g}$, *i.e.* via STE in the qth iteration as,

$$\frac{\partial \ell^q}{\partial \boldsymbol{w}_g^q} = \frac{\partial \ell^q}{\partial \hat{\boldsymbol{w}}_g^q} \tag{31}$$

Then the first and second momentums in AMSGrad are updated with $\frac{\partial \ell^q}{\partial \boldsymbol{w}_g^q}$. Accordingly, the loss increment around $\boldsymbol{w}_g^q$ is modeled as,

$$f_{ste}^q = (\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{w}_g - \boldsymbol{w}_g^q) + \frac{1}{2}(\boldsymbol{w}_g - \boldsymbol{w}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{w}_g - \boldsymbol{w}_g^q) \tag{32}$$

Since $\boldsymbol{w}_g$ is full precision, $\boldsymbol{w}_g^{q+1}$ can be directly obtained through the above AMSGrad step without projection updating,

$$\boldsymbol{w}_g^{q+1} = \boldsymbol{w}_g^q - (\boldsymbol{H}^q)^{-1}\boldsymbol{g}^q = \boldsymbol{w}_g^q - a^q\boldsymbol{m}^q/\sqrt{\hat{\boldsymbol{v}}^q} \tag{33}$$

For more details about the notations, please refer to Sec. 3.1. Similarly, the loss increment caused by $\boldsymbol{B}_g$ (see Eq.(17) and Eq.(18)) is formulated as,

$$\begin{aligned} f_{ste,\boldsymbol{B}}^q = (\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q) + \\ \frac{1}{2}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \boldsymbol{w}_g^q) \end{aligned} \tag{34}$$

Thus, the jth row in $\boldsymbol{B}_g^{q+1}$ is updated by,

$$\boldsymbol{B}_{g,j}^{q+1} = \underset{\boldsymbol{B}_{g,j}}{\operatorname{argmin}} \|\boldsymbol{B}_{g,j}\boldsymbol{\alpha}_g^q - (w_{g,j}^q - g_j^q/H_{jj}^q)\| \tag{35}$$

In addition, the speedup step (see Eq.(22) and Eq.(23)) is,

$$\begin{aligned} \boldsymbol{\alpha}_g^{q+1} = -((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}\boldsymbol{H}^q\boldsymbol{B}_g^{q+1})^{-1} \times \\ ((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}(\boldsymbol{g}^q - \boldsymbol{H}^q\boldsymbol{w}_g^q)) \end{aligned} \tag{36}$$

So far, the quantized weights are updated in a loss-aware manner as,

$$\hat{\boldsymbol{w}}_g^{q+1} = \boldsymbol{B}_g^{q+1}\boldsymbol{\alpha}_g^{q+1} \tag{37}$$

Algorithm 6: STE with Loss-aware

Input: Q, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, Training Data
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$
for $q \leftarrow 1$ **to** Q **do**
  for $l \leftarrow 1$ **to** L **do**
    Update $\hat{\boldsymbol{w}}_{l,g}^q = \boldsymbol{B}_{l,g}^q\boldsymbol{\alpha}_{l,g}^q$;
    Forward propagate convolution ;
  Compute the loss ℓ^q ;
  for $l \leftarrow L$ **to** 1 **do**
    Backward propagate gradient $\frac{\partial \ell^q}{\partial \hat{\boldsymbol{w}}_{l,g}^q}$;
    Directly approximate $\frac{\partial \ell^q}{\partial \boldsymbol{w}_{l,g}^q}$ **with** $\frac{\partial \ell^q}{\partial \hat{\boldsymbol{w}}_{l,g}^q}$;
    Update momentums of AMSGrad ;
    for $g \leftarrow 1$ **to** G_l **do**
      Update $\boldsymbol{w}_{l,g}^{q+1}$ **with** Eq.(33) ;
      Compute all values of Eq.(20) ;
      for $j \leftarrow 1$ **to** n_l **do**
        Update $\boldsymbol{B}_{l,g,j}^{q+1}$ **according to the nearest value** (**see** Eq.(35)) ;
      Update $\boldsymbol{\alpha}_{l,g}^{q+1}$ **with** Eq.(36) ;

C. Implementation Details

C.1. LeNet5 on MNIST

The MNIST dataset [22] consists of 28×28 gray scale images from 10 digit classes. We use 50000 samples in the training set for training, the rest 10000 for validation, and the 10000 samples in the test set for testing. The test accuracy is reported, when the validation dataset has the highest top-1 accuracy. We use a mini-batch with size of 128. The used LeNet5 is a modified version of [21]. For data preprocessing, we use the official example provided by Pytorch [30]. We use the default hyperparameters proposed in [32] to train LeNet5 for 100 epochs as the baseline of full precision version.

The network architecture is presented as,
20C5 - MP2 - 50C5 - MP2 - 500FC - 10SVM.

The structures of each layer chosen for ALQ are *kernel-wise, kernel-wise, subchannel-wise(2), channel-wise* respectively. The *subchannel-wise(2)* structure means all input channels are sliced into two groups with the same size, *i.e.* the group size here is $800/2 = 400$. After each pruning, the network is retrained to recover the accuracy degradation with 20 epochs of optimizing $\boldsymbol{B}_g$ and 10 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 80%, and 4 times pruning (Step 1) are executed after initialization in the reported experiment (Table 2). In the end, *i.e.* after the retraining of the last pruning step, we add another 50 epochs of optimizing steps (Sec. 3.3) to further increase the final accuracy (also applied

in the following experiments of VGG and ResNet18/34).

ALQ can fast converge in the training. However, we observe that even after the convergence, the accuracy still continue increasing slowly along the training, which is similar as the behavior of STE-based optimizer. During the retraining after each pruning step, as long as the training loss is (almost) converged with a few epochs, we can further proceed the next pruning step. We have tested that the final accuracy level is approximately the same whether we add plenty of epochs each time to slowly recover the accuracy to the original level or not. Thus, we choose a fixed modest number of retraining epochs after each pruning step to save the overall training time. In fact, this benefits from the feature of ALQ, which leverages the true gradient (w.r.t. the loss) to result a fast and stable convergence. The final added plenty of training epochs aim to further slowly regain the final accuracy level, and we use a gradually decayed learning rate in this process, *e.g.* 10^{-4} decays with 0.98 in each epoch.

C.2. VGG on CIFAR10

The CIFAR-10 dataset [19] consists of 60000 32×32 color images in 10 object classes. We use 45000 samples in the training set for training, the rest 5000 for validation, and the 10000 samples in the test set for testing. We use a mini-batch with size of 128. The used VGG net is a modified version of the original VGG [40]. For data preprocessing, we use the setting provided by [33]. We use the default Adam optimizer provided by Pytorch [32] to train full precision parameters for 200 epochs as the baseline of the full precision version. The initial learning rate is 0.01, and it decays with 0.2 every 30 epochs.

The network architecture is presented as,
2×128C3 - MP2 - 2×256C3 - MP2 - 2×512C3 - MP2 - 2×1024FC - 10SVM.

The structures of each layer chosen for ALQ are *channel-wise, pixel-wise, pixel-wise, pixel-wise, pixel-wise, pixel-wise, subchannel-wise(16), subchannel-wise(2), subchannel-wise(2)* respectively. After each pruning, the network is retrained to recover the accuracy degradation with 20 epochs of optimizing $\boldsymbol{B}_g$ and 10 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 40%, and 5/6 times pruning (Step 1) are executed after initialization in the reported experiment (Table 3).

In order to demonstrate the convergence of ALQ statistically, we plot the train loss curves (the mean of cross-entropy loss) of quantizing VGG on CIFAR10 with ALQ in 5 successive trials (see Fig. 4a). We also plot one of them with detailed training steps of ALQ (see Fig. 4b).

C.3. ResNet18/34 on ILSVRC12

The ImageNet (ILSVRC12) dataset [38] consists of 1.2 million high-resolution images for classifying in 1000 object classes. The validation set contains 50k images, which are used to report the accuracy level. We use mini-batch with size of 256. The used ResNet18/34 is from [11]. For data preprocessing, we use the setting provided by [34]. We use the ResNet18/34 provided by Pytorch [32] as the baseline of full precision version. The network architecture is the same as "resnet18/resnet34" in [35].

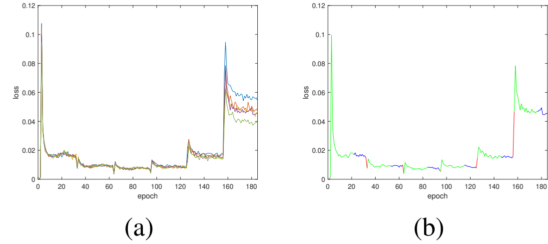

Figure 4. The train loss of VGG on CIFAR10 trained by ALQ. (a) The train loss of 5 trials. (b) A detailed example train loss. 'Magenta' stands for initialization; 'Green' stands for optimizing $\boldsymbol{B}_g$ with speedup; 'Blue' stands for optimizing $\boldsymbol{\alpha}_g$; 'Red' stands for pruning in $\boldsymbol{\alpha}$ domain. Please see this figure in color.

The structures of each layer chosen for ALQ are all *pixel-wise* except for the first layer (*kernel-wise*) and the last layer (*subchannel-wise(2)*). After each pruning, the network is retrained to recover the accuracy degradation with 10 epochs of optimizing $\boldsymbol{B}_g$ and 5 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 15%, and 5/9 times pruning (Step 1) are executed after initialization in the reported experiments (Table 4).

For quantizing a large network with an average low bitwidth (*e.g.* ≤ 2.0), we find that adding our *STE with loss-aware* steps in the end can result an around $1\% \sim 2\%$ higher accuracy (see Table 4) than adding the optimizing steps of Sec. 3.3. Thus, we add another 50 epochs of *STE with loss-aware* in the end for quantizing ResNet18/34. The learning rate is 10^{-4}, and gradually decays with 0.98 per epoch. Here, *STE with loss-aware* is just used in the end to further seeking a higher final accuracy.

We think this is due to the fact that several layers have already been pruned to an extremely low bitwidth (< 1.0-bit). With such an extremely low bitwidth, maintained full precision weights help to calibrate some aggressive steps of quantization, which slowly converges to a local optimum with a higher accuracy for a large network. Recall that maintaining full precision parameters means STE is required to approximate the gradients, since the true-gradients only relate to the quantized parameters used in the forward propagation. However, for the quantization bitwidth higher than two (> 2.0-bit), the quantizer can already take smooth steps, and the gradient approximation brought from STE damages the training process inevitably. Thus in this case, the true-gradient optimizer (our optimization steps in Sec. 3.3) can converge to a better local optimum, faster and more stable.