Adaptive Loss-aware Quantization for Multi-bit Networks

Zhongnan Qu, Zimu Zhou*, Yun Cheng, Lothar Thiele
Computer Engineering Group, ETH Zurich, Switzerland
quz@ethz.ch, zzhou@tik.ee.ethz.ch, chengyu@ethz.ch, thiele@ethz.ch

Abstract

We investigate the compression of deep neural networks by quantizing their weights and activations into multiple binary bases, known as multi-bit networks (MBNs), which accelerates the inference and reduces the storage for deployment on low-resource mobile and embedded platforms. We propose Adaptive Loss-aware Quantization (ALQ), a new MBN quantization pipeline that is able to achieve an average bitwidth below one bit without notable loss in inference accuracy. Unlike previous MBN quantization solutions that train a quantizer by minimizing the error to reconstruct full precision weights, ALQ directly minimizes the quantization-induced error on the loss function involving neither gradient approximation nor full precision calculations. ALQ also exploits strategies including adaptive bitwidth, smooth bitwidth reduction, and iterative trained quantization to allow a smaller network size without loss in accuracy. Experiment results on popular image datasets show that ALQ outperforms state-of-the-art compressed networks in terms of both storage and accuracy.

1. Introduction

There is a growing interest to deploy deep neural networks on resource-constrained devices to enable new intelligent services such as mobile assistants, augmented reality, and autonomous cars. However, deep neural networks are notoriously resource-intensive. Their complexity needs to be trimmed down to fit in mobile and embedded devices for real-time applications.

To take advantage of the various pretrained models for efficient inference on resource-constrained devices, it is common to compress the pretrained models via pruning [10], quantization [8, 9, 26, 43, 44], distillation [12], among others. We focus on quantization, especially quantizing both the full precision weights and activations of a deep neural network into binary encodes and the corresponding scaling factors [4, 36], which are also interpreted as binary basis vectors and floating-point coordinates in a geometry viewpoint [9]. Neural networks quantized with binary encodes replace expensive floating-point operations by bitwise operations, which are supported even by microprocessors and often result in small memory footprints [29]. Since the space spanned by only one-bit binary basis and one coordinate is too sparse to optimize, many researchers suggest a multi-bit network (MBN) [8, 9, 15, 26, 43, 44], which allows to obtain a small size without notable accuracy loss and still leverages bitwise operations. An MBN is usually obtained via trained quantization. Recent studies [31] leverage bit-packing and bitwise computations for efficient deploying binary networks on a wide range of general devices, which also provides more flexibility to design multi-bit/binary networks.

Existing MBN quantization schemes [8, 9, 15, 26, 43, 44] usually first predetermine a global bitwidth, and then learn a quantizer to transform the full precision parameters into binary bases and coordinates such that the quantized MBNs approximate the full precision ones without significantly decreasing inference accuracy. But their approaches suffer from the following drawbacks:

- A global bitwidth may lead to sub-optimal quantization performance. Recent studies on fixed-point quantization [18, 25] show that the optimal parameter bitwidth varies across layers.
- Previous efforts [43, 26, 44] retain network accuracy by minimizing the weight reconstruction error rather than the loss function. Such an indirect optimization objective may lead to a notable loss in accuracy. Even worse, to propagate gradients through quantization functions, all of them rely on approximated gradients, *e.g*. straight-through estimators (STE) during training.
- Many methods [36, 44] keep the first and last layer in full precision to avoid dramatic accuracy degradation [41, 28]. However, this leads to a significant storage overhead compared to the intermediate quantized layers, especially for a low bitwidth (see Sec. 5.4.3). Also, floating-point computations in these two layers can take up the majority of computation in a quantized network [27].

*Corresponding Author: Zimu Zhou.

We overcome the above drawbacks via a novel *A*daptive *L*oss-aware *Q*uantization scheme (ALQ). Instead of using a uniform bitwidth for the entire network, ALQ assigns a different bitwidth to each group of weights. More importantly, ALQ directly minimizes the loss function w.r.t. the quantized weights, by iteratively learning a quantizer that *(i)* smoothly reduces the number of binary bases and *(ii)* alternatively optimizes the remained binary bases and the corresponding coordinates. Despite prior proposals on loss-aware quantization for binary and ternary networks [14, 13, 47], they are inapplicable to MBNs due to the extended optimization space. They also need STE as approximated gradients during training. ALQ is the first loss-aware quantization scheme for MBNs and eliminates the need for approximating gradients and retaining full precision weights. ALQ is also able to quantize the first and last layers without incurring notable accuracy loss.

The main contributions of this work are as follows.

- We design ALQ, the first loss-aware quantization scheme for multi-bit networks. To the best of our knowledge, it is also the first trained quantizer without gradient approximation and maintenance of full precision parameters. ALQ also applies techniques such as progressive bitwidth reduction and iterative trained quantization to better trade off model accuracy and model size.

- ALQ enables extremely low-bit (yet dense tensor form) binary networks with an average bitwidth smaller than 1 bit. It is also the first quantization scheme that realizes adaptive bitwidth for MBNs (including the first and last layers). Experiments on CIFAR10 show that ALQ can compress VGG to an average bitwidth of 0.4-bit, but yields a significantly higher accuracy than other binary networks [36, 4].

2. Related Work

ALQ follows the trend to quantize deep neural networks using discrete bases to reduce expensive floating-point operations. Commonly used bases include fixed-point [48], power of two [16, 46], and $\{-1, 0, +1\}$ [4, 36].

We focus on quantization with binary bases *i.e.* $\{-1, +1\}$ among others for the following considerations. *(i)* If both weights and activations are quantized with the same binary basis, it is possible to evaluate 32 multiplieraccumulator operations (MACs) with only 3 instructions on a 32-bit microprocessor, *i.e.* bitwise `xnor`, `popcount`, and accumulation. This will significantly speed up the convolution operations [16]. *(ii)* A network quantized to fixed-point requires specialized integer arithmetic units (with various bitwidth) for efficient computing [1, 18], whereas a network quantized with multiple binary bases adopts the same operations mentioned before as binary networks. Popular networks quantized with binary bases include *Binary Networks* and *Multi-bit Networks*.

2.1. Quantization for Binary Networks

BNN [4] is the first network with both binarized weights and activations. It dramatically reduces the memory and computation of a neural network but often with notable accuracy loss. To resume the accuracy degradation from binarization, XNOR-Net [36] introduces a layer-wise full precision scaling factor into BNN. However, XNOR-Net leaves the first and last layers unquantized, thus consumes more memory on the disk. SYQ [6] studies the efficiency of different structures during binarization/ternarization, in terms of memory, computation and deployment. LAB [14] is the first loss-aware quantization scheme which optimizes the weights by directly minimizing the loss function.

ALQ is inspired by recent loss-aware binary networks such as LAB [14]. Loss-aware quantization has also been extended to fixed-point networks in [13]. However, existing loss-aware quantization schemes [14, 13] are inapplicable for MBNs. This is because multiple binary bases extend the optimization space with the same bitwidth (*i.e.* an optimal set of binary bases rather than a single basis), which turns to a combinatorial optimization. Other previous proposals [14, 13, 47] still require full-precision weights and gradient approximation (backward STE and forward loss-aware projection), introducing undesirable errors when minimizing the loss. In contrast, ALQ is free from gradient approximation.

2.2. Quantization for Multi-bit Networks

MBNs donate networks that use multiple binary bases to trade off computation and accuracy. Gong *et al.* propose a residual quantization process, which greedily searches the next binary basis by minimizing the residual reconstruction error [8]. Guo *et al.* improve the greedy search with a least square refinement [9]. Xu *et al.* [43] separate this search into two alternating steps, fixing coordinates then exhausted searching for optimal bases, and fixing the bases then refining the coordinates using the method in [9]. LQ-Net [44] extends the scheme of [43] with a moving average updating, which jointly quantizes weights and activations. However, similar to XNOR-Net [36], LQ-Net [44] does not quantize the first and last layers. ABC-Net [26] leverages the statistical information of all weights to construct the binary bases as a whole for all layers.

All the state-of-the-art MBN quantization schemes minimize the weight reconstruction error rather than the loss function of the network. They also rely on gradient approximation such as STEs when back propagating the quantization function. In addition, they all predetermine a uniform bitwidth for all parameters. The indirect objective, the approximated gradient, and the global bitwidth lead to sub-optimal quantization. ALQ is the first scheme to explicitly optimize the loss function and incrementally train an adaptive bitwidth while without gradient approximation.

3. Adaptive Loss-Aware Quantization

3.1. Weight Quantization Overview

Notations. As mentioned before, we aim at MBN quantization with an adaptive bitwidth. To allow adaptive bitwidth, we structure the weights in *disjoint groups*. Specifically, for the vectorized weights $\boldsymbol{w}$ of a given layer l, where $\boldsymbol{w} \in \mathbb{R}^{N\times 1}$, we divide $\boldsymbol{w}$ into G disjoint groups. For simplicity, we omit the subscript l. Each group of weights is denoted by $\boldsymbol{w}_g$, where $\boldsymbol{w}_g \in \mathbb{R}^{n\times 1}$ and $N = n \times G$. Then the quantized weights of each group $\hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g$. $\boldsymbol{\beta}_i \in \{-1, +1\}^{n\times 1}$ and $\alpha_i \in \mathbb{R}_+$ are the i^{th} binary basis and the corresponding coordinate; I_g represents the bitwidth, *i.e.* the number of binary bases, of group g. $\boldsymbol{B}_g \in \{-1, +1\}^{n\times I_g}$ and $\boldsymbol{\alpha}_g \in \mathbb{R}_+^{I_g\times 1}$ are the matrix forms of the binary bases and the coordinates. We further denote $\boldsymbol{\alpha}$ as vectorized coordinates $\{\boldsymbol{\alpha}_g\}_{g=1}^{G}$, and $\boldsymbol{B}$ as concatenated binary bases $\{\boldsymbol{B}_g\}_{g=1}^{G}$ of all the weight groups in layer l. A layer l quantized as above yields an average bitwidth $I = \frac{1}{G}\sum_{g=1}^{G} I_g$. We discuss the choice of group size n, and the initial $\boldsymbol{B}_g$, $\boldsymbol{\alpha}_g$, I_g in Sec. 5.1.

Problem Formulation. ALQ quantizes weights to minimize the loss function rather than the reconstruction error. Hence we formulate the following optimization problem for layer l.

$$\min_{\hat{\boldsymbol{w}}_g} \quad \ell\left(\hat{\boldsymbol{w}}_g\right) \tag{1}$$

$$\text{s.t.} \quad \hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g \tag{2}$$

$$\text{card}(\boldsymbol{\alpha}) = I \times G \leq \text{I}_{\text{min}} \times G \tag{3}$$

where ℓ is the loss; $\text{card}(.)$ denotes the cardinality of the set, here, the total number of elements in $\boldsymbol{\alpha}$; I_{min} is the desired average bitwidth.

ALQ tries to solve the optimization problem in Eq.(1)-Eq.(3) by converting it into two subproblems and solving them *iteratively* in two separated steps (see the whole pipeline in Alg. 5 in Appendix B.3).

- **Step 1: Pruning in α Domain** (Sec. 3.2). In this step, we progressively reduce the average bitwidth I for a layer l by pruning the least important (w.r.t. the loss) coordinates in $\boldsymbol{\alpha}$ domain. Note that removing an element α_i will also lead to the removal of the binary basis β_i, which in effect results in a smaller bitwidth I_g for group g. This way, no sparse tensor is introduced. Sparse tensors could lead to a detrimental irregular computation. Since the importance of each weight group differs, the resulting I_g varies across groups, and thus contributes to an adaptive bitwidth I_g for each group. In this step, we only set some elements of $\boldsymbol{\alpha}$ to zero (also remove them from $\boldsymbol{\alpha}$ leading to a reduced I_g) without changing the others. The optimization problem for Step 1 is:

$$\min_{\boldsymbol{\alpha}} \quad \ell\left(\boldsymbol{\alpha}\right) \tag{4}$$

$$\text{s.t.} \quad \text{card}(\boldsymbol{\alpha}) \leq \text{I}_{\text{min}} \times G \tag{5}$$

- **Step 2: Optimizing Binary Bases $\boldsymbol{B}_g$ and Coordinates $\boldsymbol{\alpha}_g$** (Sec. 3.3). In this step, we retrain the remaining binary bases and coordinates to recover the accuracy degradation induced by the bitwidth reduction. Similar to [43], we take an alternative approach for better accuracy recovery. Specifically, we first search for a new set of binary bases w.r.t. the loss given fixed coordinates. Then we optimize the coordinates by fixing the binary bases. The optimization problem for Step 2 is:

$$\min_{\hat{\boldsymbol{w}}_g} \quad \ell\left(\hat{\boldsymbol{w}}_g\right) \tag{6}$$

$$\text{s.t.} \quad \hat{\boldsymbol{w}}_g = \sum_{i=1}^{I_g} \alpha_i \boldsymbol{\beta}_i = \boldsymbol{B}_g \boldsymbol{\alpha}_g \tag{7}$$

Optimizer Framework. Previous STE based methods project the full precision parameters onto quantization bases during forward, and update the full precision parameters with the approximated gradient (from quantized values) during backward. Here, we consider both subproblems in the above two steps as an optimization problem with *domain constraints*. We propose to solve them using a same optimizer framework such as subgradient methods with projection update [5].

In our case, the optimization problem in Eq.(6)-Eq.(7) imposes domain constraints on $\boldsymbol{B}_g$ because they can only be discrete binary bases. The optimization problem in Eq.(4)-Eq.(5) can also be considered as with a trivial domain constraint that the output $\boldsymbol{\alpha}$ should be a subset (subvector) of the input $\boldsymbol{\alpha}$. Furthermore, the feasible sets for both $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}$ are bounded.

Subgradient methods with projection update are effective to solve problems in the form of $\min_{\boldsymbol{x}}(\ell(\boldsymbol{x}))$ s.t. $\boldsymbol{x} \in \mathbb{X}$ [5]. We apply AMSGrad [37], an adaptive stochastic subgradient method with projection update, as the common optimizer framework in the two steps. At iteration s, AMSGrad generates the next update as

$$\begin{aligned} \boldsymbol{x}^{s+1} &= \Pi_{\mathbb{X},\sqrt{\hat{\boldsymbol{V}}^s}}(\boldsymbol{x}^s - a^s \boldsymbol{m}^s/\sqrt{\hat{\boldsymbol{v}}^s}) \\ &= \underset{\boldsymbol{x}\in\mathbb{X}}{\text{argmin}} \, \|(\sqrt{\hat{\boldsymbol{V}}^s})^{1/2}(\boldsymbol{x} - (\boldsymbol{x}^s - \frac{a^s \boldsymbol{m}^s}{\sqrt{\hat{\boldsymbol{v}}^s}}))\| \end{aligned} \tag{8}$$

where Π is projection operator; $\mathbb{X}$ is the feasible domain of $\boldsymbol{x}$; a^s is the learning rate; $\boldsymbol{m}^s$ is the (unbiased) first momentum;

$\hat{\boldsymbol{v}}^s$ is the (unbiased) maximum second momentum; and $\hat{\boldsymbol{V}}^s$ is the diagonal matrix of $\hat{\boldsymbol{v}}^s$.

In our context, Eq.(8) can be written as,

$$\hat{\boldsymbol{w}}_g^{s+1} = \underset{\hat{\boldsymbol{w}}_g \in \mathbb{F}}{\operatorname{argmin}} f^s(\hat{\boldsymbol{w}}_g) \tag{9}$$

$$f^s = (a^s \boldsymbol{m}^s)^{\mathrm{T}}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s) + \frac{1}{2}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s)^{\mathrm{T}}\sqrt{\hat{\boldsymbol{V}}^s}(\hat{\boldsymbol{w}}_g - \hat{\boldsymbol{w}}_g^s) \tag{10}$$

where $\mathbb{F}$ is the feasible domain of $\hat{\boldsymbol{w}}_g$.

Step 1 and Step 2 have different feasible domain of $\mathbb{F}$ according to their objective (details in Sec. 3.2 and Sec. 3.3). Eq.(10) approximates the loss increment incurred by $\hat{\boldsymbol{w}}_g$ around the current point $\hat{\boldsymbol{w}}_g^s$ as a quadratic model function under domain constraints [5, 37]. For simplicity, we replace $a^s \boldsymbol{m}^s$ with $\boldsymbol{g}^s$ and replace $\sqrt{\hat{\boldsymbol{V}}^s}$ with $\boldsymbol{H}^s$. $\boldsymbol{g}^s$ and $\boldsymbol{H}^s$ are iteratively updated by the loss gradient of $\hat{\boldsymbol{w}}_g^s$. Thus, the required input of each AMSGrad step is $\frac{\partial \ell^s}{\partial \hat{\boldsymbol{w}}_g^s}$. Since $\hat{\boldsymbol{w}}_g^s$ is used as an intermediate value during the forward, it can be directly obtained during the backward.

3.2. Pruning in α Domain

As introduced in Sec. 3.1, we reduce the bitwidth I by pruning the elements in $\boldsymbol{\alpha}$ w.r.t. the resulting loss. If one element α_i in $\boldsymbol{\alpha}$ is pruned, the corresponding dimension is also removed from $\boldsymbol{B}$. Now we explain how to instantiate the optimizer in Eq.(9) to solve the problem in Eq.(4)-Eq.(5) of Step 1.

The cardinality of the chosen subset (*i.e.* the average bitwidth) is uniformly reduced over the iterations. For example, assuming that there are T iterations in total, the initial bitwidth is I^0 and the desired bitwidth after T iterations I^T is $\mathrm{I_{min}}$. Then at each iteration t, the equal number ($M_p = \mathrm{round}((I^0 - \mathrm{I_{min}}) \times G/T)$) of α_i^t is pruned in this layer. This way, the cardinality after T iterations will be smaller than $\mathrm{I_{min}} \times G$. For more details, please see the pseudocode Alg. 2 in Appendix B.1.

When pruning in the $\boldsymbol{\alpha}$ domain, $\boldsymbol{B}$ is regarded as invariant. Hence Eq.(9) and Eq.(10) become

$$\boldsymbol{\alpha}^{t+1} = \underset{\boldsymbol{\alpha} \in \mathbb{P}}{\operatorname{argmin}}\ f_{\boldsymbol{\alpha}}^t(\boldsymbol{\alpha}) \tag{11}$$

$$f_{\boldsymbol{\alpha}}^t = (\boldsymbol{g}_{\boldsymbol{\alpha}}^t)^{\mathrm{T}}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^t) + \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^t)^{\mathrm{T}}\boldsymbol{H}_{\boldsymbol{\alpha}}^t(\boldsymbol{\alpha} - \boldsymbol{\alpha}^t) \tag{12}$$

where $\boldsymbol{g}_{\boldsymbol{\alpha}}^t$ and $\boldsymbol{H}_{\boldsymbol{\alpha}}^t$ are similar as in Eq.(10) but are in the $\boldsymbol{\alpha}$ domain. If α_i^t is pruned, the i^{th} element in $\boldsymbol{\alpha}$ is set to 0 in the above Eq.(11) and Eq.(12). Therefore, the constrained domain $\mathbb{P}$ is taken as all possible vectors with M_p zero elements in $\boldsymbol{\alpha}^t$.

AMSGrad uses a diagonal matrix of $\boldsymbol{H}_{\boldsymbol{\alpha}}^t$ in the quadratic model function, which decouples each element in $\boldsymbol{\alpha}^t$. This means the loss increment caused by several α_i^t equals the sum of the increments caused by them individually, which is calculated as:

$$f_{\boldsymbol{\alpha},i}^t = -g_{\boldsymbol{\alpha},i}^t\ \alpha_i^t + \frac{1}{2}\ H_{\boldsymbol{\alpha},ii}^t\ (\alpha_i^t)^2 \tag{13}$$

All items of $f_{\boldsymbol{\alpha},i}^t$ are sorted in ascending. Then the first M_p items (α_i^t) in the sorted list are removed from $\boldsymbol{\alpha}^t$, and results in a smaller cardinality $I^t \times G$. The input of the AMSGrad step in $\boldsymbol{\alpha}$ domain is the loss gradient of $\boldsymbol{\alpha}_g^t$, which can be computed with the chain rule,

$$\frac{\partial \ell^t}{\partial \boldsymbol{\alpha}_g^t} = \boldsymbol{B}_g^{t\ \mathrm{T}} \frac{\partial \ell^t}{\partial \hat{\boldsymbol{w}}_g^t} \tag{14}$$

$$\hat{\boldsymbol{w}}_g^t = \boldsymbol{B}_g^t \boldsymbol{\alpha}_g^t \tag{15}$$

Our pipeline allows to reduce the bitwidth smoothly, since the average bitwidth can be floating-point. In ALQ, since different layers have a similar group size (see Sec. 5.1), the loss increment caused by pruning is sorted among all layers, such that only a global pruning number needs to be determined. This step not only provides a loss-aware adaptive bitwidth, but also seeks a better initialization for training the following lower bitwidth quantization, since quantized weights may be relatively far from their original full precision values.

3.3. Optimizing Binary Bases and Coordinates

After pruning, the loss degradation needs to be recovered. Following Eq.(9), the objective in Step 2 is

$$\hat{\boldsymbol{w}}_g^{s+1} = \underset{\hat{\boldsymbol{w}}_g \in \mathbb{F}}{\operatorname{argmin}}\ f^s(\hat{\boldsymbol{w}}_g) \tag{16}$$

The constrained domain $\mathbb{F}$ is decided by both binary bases and real valued coordinates. Hence directly searching optimal $\hat{\boldsymbol{w}}_g$ is NP-hard. Instead, we optimize $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}_g$ in an alternative manner, as with prior MBN quantization w.r.t. the reconstruction error [43, 44].

Optimizing $\boldsymbol{B}_g$. We directly search the optimal bases with AMSGrad. In each optimization step q, we fix $\boldsymbol{\alpha}_g^q$, and update $\boldsymbol{B}_g^q$. Also, we find the optimal increment for each group of weights, such that it converts to a new set of binary bases, $\boldsymbol{B}_g^{q+1}$. This optimization step searches a new space spanned by $\boldsymbol{B}_g^{q+1}$ according to the loss reduction, which prevents the pruned space to be always a subspace of the previous one. For more details, please see the pseudocode Alg. 3 in Appendix B.2.1.

According to Eq.(9) and Eq.(10), the optimal $\boldsymbol{B}_g$ w.r.t. the loss is updated by,

$$\boldsymbol{B}_g^{q+1} = \underset{\boldsymbol{B}_g \in \{-1,+1\}^{n \times I_g}}{\operatorname{argmin}}\ f^q(\boldsymbol{B}_g) \tag{17}$$

$$\begin{aligned} f^q = &(\boldsymbol{g}^q)^{\mathrm{T}}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q) + \\ &\frac{1}{2}(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q)^{\mathrm{T}}\boldsymbol{H}^q(\boldsymbol{B}_g\boldsymbol{\alpha}_g^q - \hat{\boldsymbol{w}}_g^q) \end{aligned} \tag{18}$$

where $\hat{\boldsymbol{w}}_g^q = \boldsymbol{B}_g^q \boldsymbol{\alpha}_g^q$.

Since $\boldsymbol{H}^q$ is diagonal, each row vector in $\boldsymbol{B}_g^{q+1}$ can be independently determined. For example, the j^{th} row is computed as,

$$\boldsymbol{B}_{g,j}^{q+1} = \underset{\boldsymbol{B}_{g,j}}{\operatorname{argmin}} \|\boldsymbol{B}_{g,j}\boldsymbol{\alpha}_g^q - (\hat{w}_{g,j}^q - g_j^q / H_{jj}^q)\| \tag{19}$$

In general, $n >> I_g$. For each group, we firstly compute all 2^{I_g} possible values of

$$\boldsymbol{b}^{\mathrm{T}}\boldsymbol{\alpha}_g^q\ , \quad \boldsymbol{b}^{\mathrm{T}} \in \{-1, +1\}^{1\times I_g} \tag{20}$$

Then each row vector $\boldsymbol{B}_{g,j}$ can be directly assigned to the optimal $\boldsymbol{b}^{\mathrm{T}}$ through exhaustive search.

Optimizing $\boldsymbol{\alpha}_g$. The above obtained set of binary bases $\boldsymbol{B}_g$ spans a new linear space. Current $\boldsymbol{\alpha}_g$ unlikely happens to be a (local) optimal point w.r.t. the loss in this space, so now we optimize $\boldsymbol{\alpha}_g$. Since $\boldsymbol{\alpha}_g$ is full precision, *i.e.* $\boldsymbol{\alpha}_g \in \mathbb{R}^{I_g\times 1}$, there is no domain constraint and thus no need for projection updating. Optimizing full precision $\boldsymbol{w}_g$ takes incremental steps in original n-dim full space (spanned by orthonormal bases). Similarly, optimizing $\boldsymbol{\alpha}_g$ searches steps in a I_g-dim subspace (spanned by $\boldsymbol{B}_g$). This means, the conventional training strategy can be directly applied to optimize $\boldsymbol{\alpha}_g$ (see Alg. 4 in Appendix B.2.2).

Similar as Eq.(11) and Eq.(12), we construct an AMSGrad optimizer in $\boldsymbol{\alpha}$ domain but without projection updating, for each group in the p^{th} step as,

$$\boldsymbol{\alpha}_g^{p+1} = \boldsymbol{\alpha}_g^p - a_{\boldsymbol{\alpha}}^p \boldsymbol{m}_{\boldsymbol{\alpha}}^p / \sqrt{\hat{\boldsymbol{v}}_{\boldsymbol{\alpha}}^p} \tag{21}$$

We also add an L2-norm regularization on $\boldsymbol{\alpha}_g$ to enforce the unimportant coordinates to zero. If there exists a negative value in $\boldsymbol{\alpha}_g$, the corresponding basis is set to its negative complement, to keep $\boldsymbol{\alpha}_g$ semi-positive definite. Optimizing $\boldsymbol{B}_g$ and $\boldsymbol{\alpha}_g$ does not influence the bitwidth (the number of binary bases) I_g.

Optimization Speedup. Since $\boldsymbol{\alpha}_g$ is full precision, updating $\boldsymbol{\alpha}_g^q$ is much cheaper than exhaustively search $\boldsymbol{B}_g^{q+1}$. Even if the main purpose of the first step in Sec. 3.3 is optimizing bases, we also add an updating process for $\boldsymbol{\alpha}_g^q$ in each optimization step q.

We fix $\boldsymbol{B}_g^{q+1}$, and update $\boldsymbol{\alpha}_g^q$. The overall increment of quantized weights from both updating steps is,

$$\hat{\boldsymbol{w}}_g^{q+1} - \hat{\boldsymbol{w}}_g^q = \boldsymbol{B}_g^{q+1}\boldsymbol{\alpha}_g^{q+1} - \boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q \tag{22}$$

Substituting Eq.(22) into Eq.(9) and Eq.(10), we have,

$$\begin{aligned}\boldsymbol{\alpha}_g^{q+1} = -((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}\boldsymbol{H}^q\boldsymbol{B}_g^{q+1})^{-1}\times \\ ((\boldsymbol{B}_g^{q+1})^{\mathrm{T}}(\boldsymbol{g}^q - \boldsymbol{H}^q\boldsymbol{B}_g^q\boldsymbol{\alpha}_g^q))\end{aligned} \tag{23}$$

To ensure the inverse in Eq.(23) exists, we add $\lambda\mathbf{I}$ onto $(\boldsymbol{B}_g^{q+1})^{\mathrm{T}}\boldsymbol{H}^q\boldsymbol{B}_g^{q+1}$, where $\lambda = 10^{-6}$.

4. Activation Quantization

To leverage bitwise operations for speedup, the inputs of each layer (*i.e.* the activation output of the last layer) also need to be quantized into the multi-bit binary form. Unlike previous works [44] that quantize activations with a different binary basis ($\{0, +1\}$) as weights, we also quantize activations with $\{-1, +1\}$. This way, we only need 3 instructions rather than 5 instructions to replace the original 32 MACs (see Sec. 2).

Our activation quantization mainly follows the idea in [2] for fixed-point activation quantization, but it is adapted to the multi-bit form. Specially, we replace ReLu with a layerwise step activation function. The vectorized activation $\boldsymbol{x}$ of the l^{th} layer is quantized as

$$\boldsymbol{x} \doteq \hat{\boldsymbol{x}} = x_{ref} + \boldsymbol{D}\boldsymbol{\gamma} \tag{24}$$

where $\boldsymbol{D} \in \{-1, +1\}^{N_x\times I_x}$, and $\boldsymbol{\gamma} \in \mathbb{R}_+^{I_x\times 1}$. N_x is the dimension of $\boldsymbol{x}$, and I_x is the quantization bitwidth for activations. Unlike weights, activations change at run-time during inference. Each activation value is quantized through a binary search among all possible 2^{I_x} values, and the corresponding row of $\boldsymbol{D}$ is encoded with the nearest value. To fit in the output range of ReLu, we introduce a layerwise (positive) floating-point reference, x_{ref}. During inference, x_{ref} is convoluted with the weights of the next layer and added to the bias. Hence the introduction of x_{ref} does not lead to extra computations. The output of the last layer is not quantized, as it does not involve computations anymore. $\boldsymbol{\gamma}$ and x_{ref} are updated with a running average to minimize the statistical reconstruction error. The (quantized) weights are also fine-tuned to resume the accuracy degradation. Here, we only set a global bitwidth for all layers in activation quantization.

5. Experiments

We implement ALQ with Pytorch [30], and evaluate its performance on MNIST [22], CIFAR10 [19], and ILSVRC12 (ImageNet) [38] using LeNet5 [21], VGG [14, 36], and ResNet18/34 [11], respectively. More implementation details are provided in Appendix C.

5.1. ALQ Initialization

We adapt the network sketching proposed in [9] for $\hat{\boldsymbol{w}}_g$ initialization, and realize a structured sketching (see Alg. 1 in Appendix A.1). The important parameters in Alg. 1 are chosen as below.

Group Size n. We empirically decide a range for the group size n by trading off between the weight reconstruction error and the storage compression rate. A group size from 32 to 512 achieves a good balance (Appendix A.2). Accordingly, for a convolution layer, grouping in channel-wise ($\boldsymbol{w}_{c,:,:,:}$), kernel-wise ($\boldsymbol{w}_{c,d,:,:}$), and pixel-wise ($\boldsymbol{w}_{c,:,h,w}$) appears to be appropriate. Channel-wise $\boldsymbol{w}_{c,:}$ and subchannel-wise $\boldsymbol{w}_{c,d:d+n}$ grouping are suited for a fully connected layer. In addition, the most frequently used structures for currently popular networks are pixel-wise (convolution layer) and channel-wise (fully connected layer), which also align with the bit-packing approach in [31].

Maximum Bitwidth $\mathrm{I}_{\max}$ for Group g. The initial I_g is controlled by a predefined initial reconstruction precision or a maximum bitwidth. We notice that the accuracy degradation caused by the initialization can be fully recovered after several optimization epochs proposed in Sec. 3.3, if the maximum bitwidth is 8. For example, ResNet18 on ILSVRC12 after such an initialization can be retrained to a Top-1/5 accuracy of 70.3%/89.4%, even higher than its full precision counterpart (69.8%/89.1%). For smaller networks, *e.g.* VGG on CIFAR10, a maximum bitwidth of 6 is sufficient.

5.2. Convergence Analysis

Settings. This experiment shows the advantages of our optimizer in Sec. 3.3 in terms of convergence. The optimizing $\boldsymbol{B}_g$ step with speedup (also Alg. 3) is compared, since it takes a similar alternating step as previous works [43, 44]. Recall that our optimizer *(i)* has no gradient approximation and *(ii)* directly minimizes the loss. We use AMSGrad[1] with a fixed learning rate of 0.001, and compare our optimizer with following baselines.

- *STE with rec. error:* This baseline quantizes the maintained full precision parameters by minimizing the reconstruction error (rather than the loss) during forward and approximates gradients via STE during backward. This approach is adopted in some of the best-performing quantization schemes such as LQ-Net [44].

- *STE with loss-aware:* This baseline approximates gradients via STE but performs a loss-aware projection (adapted from our ALQ) during forward. It can be considered as a multi-bit extension of prior loss-aware quantization schemes for binary and ternary networks [14, 13].

Results. Fig. 1 shows the Top-1 validation accuracy of different optimizers, with increasing epochs on uniform bitwidth MBNs. ALQ exhibits not only a more stable and faster convergence, but also a higher accuracy. The exception is 2-bit ResNet18. ALQ converges faster, but the validation accuracy trained with STE gradually exceeds ALQ after about 20 epochs. This is because initializing 2-bit ResNet18 is almost random: Top-1 accuracy after initialization is 0.00099 (1000-class). For training a large network from scratch with ≤ 2 bitwidth, the positive effect brought from the maintained high-precision trace can compensate some negative effect caused by the gradient approximation. In this case, keeping full precision parameters may slowly converge to a better local optimum. However, ALQ avoids such a challenge by reducing bitwidth progressively.

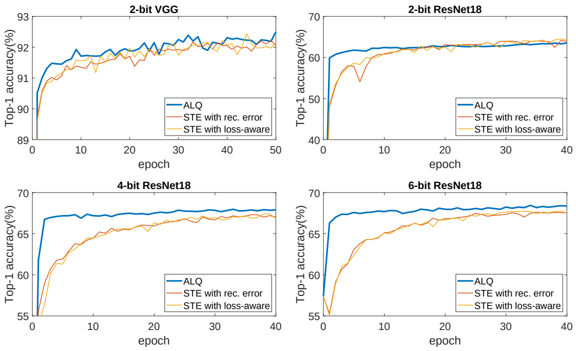


Figure 1. Validation accuracy trained with ALQ/baselines.

5.3. Effectiveness of Adaptive Bitwidth

Settings. This experiment demonstrates the performance of incrementally trained adaptive bitwidth in ALQ. Uniform bitwidth quantization (an equal bitwidth allocation across all groups in all layers) is taken as the baseline. The baseline is trained with the same number of epochs as the sum of all epochs during the bitwidth reduction (Sec. 3.2). Both ALQ and the baseline are trained with the same learning rate decay schedule.

Results. Table 1 shows that there is a large Top-1 accuracy gap between an adaptive bitwidth trained with ALQ and a uniform bitwidth (baseline). In addition to the overall average bitwidth (I_W), we also plot the distribution of the average bitwidth across layers (both models in Table 1) in Fig. 2. Generally, the first several layers and the last layer are more sensitive to the loss, thus require a higher bitwidth. The shortcut layers in ResNet architecture (*e.g.* the 8th, 13rd, 18th layers in ResNet18) also need a higher bitwidth. We think this is due to the fact that the shortcut pass helps the information forward/backward propagate through the blocks. Since the average of adaptive bitwidth can have a decimal part, ALQ can achieve a compression rate with a much higher resolution than a uniform bitwidth, which not only controls a more precise trade-off between storage and accuracy, but also benefits our incremental bitwidth reduction (pruning) scheme.

It is worth noting that the optimization and pruning in ALQ follow the same metric, *i.e.* the loss increment modeled by a quadratic function, allowing them to work in synergy. We replace the step of optimizing $\boldsymbol{B}_g$ in ALQ with a STE step (with the reconstruction forward, see in Sec. 5.2), and keep other steps unchanged in the pipeline. When reduced

[1] AMSGrad can also optimize full precision parameters.

Table 1. Comparison between Baseline (Uniform Bitwidth) and ALQ (Adaptive Bitwidth)

Method	I_W	Top-1
Baseline VGG (uniform)	1	91.8%
ALQ VGG	**0.66**	**92.0%**
Baseline ResNet18 (uniform)	2	66.2%
ALQ ResNet18	**2.0**	**68.9%**

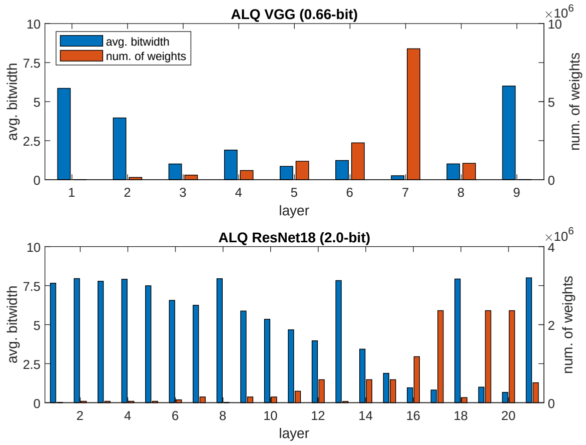


Figure 2. Distribution of the average bitwidth and the number of weights across layers.

to an average bitwidth of 0.66-bit, the simple combination of a STE step with a pruning step can only reach 90.7% Top-1 accuracy, which is significantly worse than ALQ's 92.0%.

5.4. Comparison with States-of-the-Arts

5.4.1 Non-structured Pruning on MNIST

Settings. Since ALQ can be considered as a pruning scheme in α domain, we first compare ALQ with two widely used non-structured pruning schemes: Deep Compression (DC) [10] and ADMM-Pruning (ADMM) [45], *i.e.* pruning in the original w domain. For fair comparison, we implement a modified LeNet5 model as in [10, 45] on MNIST dataset [22] and compare the Top-1 prediction accuracy and the compression rate. Note that the storage consumption only counts the weights, since the weights take the most majority of the storage (even after quantization) in comparison to others, *e.g.* bias, activation quantizer, batch normalization, *etc*. The storage consumption of weights in ALQ includes the look-up-table for the resulting I_g in each group.

Table 2. Comparison with State-of-the-Art Non-structured Pruning Methods (LeNet5 on MNIST).

Method	Weights (CR)	Top-1
FP	1720KB (1×)	99.19%
DC [10]	44.0KB (39×)	**99.26%**
ADMM [45]	24.2KB (71×)	99.20%
ALQ	**22.7KB** (**76**×)	**99.12%**

Results. ALQ shows the highest compression rate (**76**×) while keeping acceptable Top-1 accuracy compared to the two other pruning methods (see Table 2). FP stands for full precision, and the weights in the original full precision LeNet5 consume 1720KB [10]. CR donates the compression rate of storing the weights.

It is worth mentioning that both DC [10] and ADMM [45] rely on sparse tensors, which need special libraries or hardware for execution [24]. The operand (the shared quantized value) in these pruning methods is still floating-point. Hence they hardly utilize bitwise operations for speedup. In contrast, ALQ achieves a higher compression rate without sparse tensors, which is more suited for general off-the-shelf pipelined platforms.

The average bitwidth of ALQ is below 1 bit (1 bit corresponds to a compression rate slightly less than 32), which means some groups are fully removed. In fact, this process leads to a new network architecture containing less output channels of each layer, and thus the corresponding input channels of the next layers can be safely removed. The original configuration $20-50-500-10$ is now changed to $18-45-231-10$.

5.4.2 Binary Networks on CIFAR10

Settings. In this experiment, we compare the performance of ALQ with state-of-the-art binary networks [4, 36, 14]. A binary network is an MBN with the lowest bitwidth, *i.e.* single bit. Thus, the storage consumption of a binary network can be regarded as the lower bound of a (multi-bit) binary network. For fair comparison, we implement a small version of VGG from [40] on CIFAR10 dataset [19], which is widely used to report results by state-of-the-art binary networks [4, 36, 14].

Table 3. Comparison with State-of-the-Art Binary Networks (VGG on CIFAR10).

Method	I_W	Weights (CR)	Top-1
FP	32	56.09MB (1×)	92.8%
BC [3]	1	1.75MB (32×)	90.1%
BWN [36]*	1	1.82MB (31×)	90.1%
LAB [14]	1	1.77MB (32×)	89.5%
AQ [18]	0.27	1.60MB (35×)	90.9%
ALQ	**0.66**	**1.29MB (43**×)	**92.0%**
ALQ	**0.40**	**0.82MB (68**×)	**90.9%**

*: both first and last layers are unquantized.

Results. Table 3 shows the performance comparison to popular binary networks. I_W stands for the quantization bitwidth for weights. Since ALQ has an adaptive quantization bitwidth, the reported bitwidth of ALQ is an average value of all weights. For statistic information, we plot multiple training loss curves in Appendix C.2.

ALQ allows to compress the network to under 1-bit, which remarkably reduces the storage and computation. ALQ achieves the smallest weight storage and the highest accuracy compared to all weights binarization methods BC [3], BWN [36], LAB [14]. Similar to results on LeNet5, ALQ generates a new architecture with fewer output channels per layer, which further reduce our models in Table 3 to 1.01MB (0.66-bit) or even 0.62MB (0.40-bit). Besides, the computation and the run-time memory can also decrease.

In addition to binary networks, we also compare with AQ [18], the state-of-the-art adaptive fixed-point quantizer. It assigns a different bitwidth for each parameter based on its sensitivity, and also realizes an elementwise pruning for 0-bit parameters. Our ALQ not only consumes less storage, but also acquires a higher accuracy than AQ [18]. Besides, this kind of non-standard quantization bitwidth in AQ cannot efficiently run on general hardwares due to the irregularity [18], which is not the case for ALQ.

5.4.3 MBNs on ILSVRC12

Settings. We quantize both the weights and the activations of ResNet18/34 [11] with a low bitwidth ($\leq$ 2bit) on ILSVRC12 dataset [38], and compare our results with state-of-the-art multi-bit networks. The results for the full precision version is provided by Pytorch [30]. We choose ResNet18, as it is a popular model on ILSVRC12 used in previous quantization schemes. ResNet34 is a deeper network used more in recent quantization papers.

Results. Table 4 shows that ALQ obtains the highest accuracy with the smallest network size on ResNet18/34, in comparison with other weight and weight+activation quantization approaches. I_W and I_A are the quantization bitwidth for weights and activations respectively.

Several schemes (marked with *) are not able to quantize the first and last layers, since quantizing both layers with the same bitwidth as other layers will cause a huge accuracy degradation [41, 28]. It is worth noting that the first and last layers with floating-point values occupy 2.09MB storage in ResNet18/34, which is still a significant consumption under such a low bitwidth quantization. We can simply observe this enormous difference between TWN [23] and LQ-Net [44] in Table 4 for example. The evolved floating-point computations in both layers cannot be accelerated with bitwise operations either. In addition, if both weights and activations are quantized using the same binary basis across all layers, the number of bitwise operations is proportional to $I_W \times I_A$. ALQ also needs less bitwise operations related to other MBNs, *e.g.* ABC-Net [26].

6. Conclusion

In this paper, we propose a novel loss-aware trained quantizer for multi-bit network, which realizes an adaptive bitwidth for all layers (w.r.t. the loss). The experiments on current open datasets reveal that ALQ outperforms state-of-the-art multi-bit networks in both accuracy and storage. Currently, we are deploying ALQ on a mobile platform to measure the inference efficiency.

Table 4. Comparison with State-of-the-Art Multi-bit Networks (ResNet18/34 on ILSVRC12).

Method	I_W/I_A	Weights	Top-1
ResNet18			
FP [30]	32/32	46.72MB	69.8%
TWN [23]	2/32	2.97MB	61.8%
LR [39]	2/32	4.84MB	63.5%
LQ [44]*	2/32	4.91MB	68.0%
QIL [17]*	2/32	4.88MB	68.1%
INQ [46]	3/32	4.38MB	68.1%
ABC [26]	5/32	7.41MB	68.3%
ALQ	**2.0/32**	**3.44MB**	**68.9%**
BWN [36]*	1/32	3.50MB	60.8%
LR [39]*	1/32	3.48MB	59.9%
DSQ [7]*	1/32	3.48MB	63.7%
ALQ	**1.0/32**	**1.83MB**	**65.6%**
LQ [44]*	2/2	4.91MB	64.9%
PACT [2]*	2/2	4.88MB	64.4%
QIL [17]*	2/2	4.88MB	65.7%
DSQ [7]*	2/2	4.88MB	65.2%
RQ [27]	4/4	5.93MB	62.5%
ABC [26]	5/5	7.41MB	65.0%
ALQ	**2.0/2**	**3.44MB**	**66.4%**
SYQ [6]*	1/8	3.48MB	62.9%
LQ [44]*	1/2	3.50MB	62.6%
PACT [2]*	1/2	3.48MB	62.9%
ALQ	**1.0/2**	**1.83MB**	**63.2%**
ResNet34			
FP [30]	32/32	87.12MB	73.3%
LQ [44]*	2/2	7.47MB	69.8%
QIL [17]*	2/2	7.40MB	70.6%
DSQ [7]*	2/2	7.40MB	70.0%
ABC [26]	5/5	13.80MB	68.4%
ALQ	**2.0/2**	**6.43MB**	**71.1%**
TBN [41]*	1/2	4.78MB	58.2%
LQ [44]*	1/2	4.78MB	66.6%
ALQ	**1.0/2**	**3.42MB**	**67.3%**

*: both first and last layers are unquantized.

References

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394, 2017. 2

[2] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*,

abs/1805.06085, 2018. 5, 8

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of Advances in Neural Information Processing Systems*, pages 3123–3131, 2015. 7, 8

[4] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016. 1, 2, 7

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. 3, 4

[6] Julian Faraone, Nicholas J. Fraser, Michaela Blott, and Philip Heng Wai Leong. SYQ: learning symmetric quantization for efficient deep neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 4300–4309, 2018. 2, 8

[7] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of International Conference in Computer Vision*, 2019. 8

[8] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 1, 2

[9] Yiwen Guo, Anbang Yao, Hao Zhao, and Yurong Chen. Network sketching: exploiting binary structure in deep cnns. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 5955–5963, 2017. 1, 2, 5, 11

[10] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of International Conference on Learning Representations*, 2016. 1, 7

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 5, 8, 14

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *Proceedings of NIPS Deep Learning Workshop*, 2014. 1

[13] Lu Hou and James T Kwok. Loss-aware weight quantization of deep networks. In *Proceedings of International Conference on Learning Representations*, 2018. 2, 6

[14] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. In *Proceedings of International Conference on Learning Representations*, 2017. 2, 5, 6, 7, 8

[15] Qinghao Hu, Peisong Wang, and Jian Cheng. From hashing to cnns: Training binary weight networks via hashing. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2018. 1

[16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2017. 2

[17] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019. 8

[18] Soroosh Khoram and Jing Li. Adaptive quantization of neural networks. In *Proceedings of International Conference on Learning Representations*, 2018. 1, 2, 7, 8

[19] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). http://www.cs.toronto.edu/~kriz/cifar.html. 5, 7

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 11

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 5, 14

[22] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010. 5, 7

[23] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. In *Proceedings of Advances in Neural Information Processing Systems*, 2016. 8

[24] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *Proceedings of International Conference on Learning Representations*, 2017. 7

[25] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of ACM International Conference on Machine Learning*, pages 2849–2858, 2016. 1

[26] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Proceedings of Advances in Neural Information Processing Systems*, pages 345–353, 2017. 1, 2, 8

[27] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. In *Proceedings of International Conference on Learning Representations*, 2019. 1, 8

[28] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *Proceedings of International Conference on Learning Representations*, 2018. 1, 8

[29] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. WRPN: Wide reduced-precision networks. In *Proceedings of International Conference on Learning Representations*, 2018. 1

[30] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Proceedings of NIPS Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, 2017. 5, 8, 11, 14

[31] Fabrizio Pedersoli, George Tzanetakis, and Andrea Tagliasacchi. Espresso: Efficient forward propagation for binary deep neural networks. In *Proceedings of International Conference on Learning Representations*, 2018. 1, 6, 12

[32] Pytorch. Pytorch example of lenet-5 on mnist. https://github.com/pytorch/examples/blob/master/mnist/main.py. Accessed: 2019-09-28. 14

[33] Pytorch. Pytorch example on cifar10. https://github.com/kuangliu/pytorch-cifar/blob/master/main.py. Accessed: 2019-10-08. 14

[34] Pytorch. Pytorch example on imagenet. https://github.com/pytorch/examples/blob/master/imagenet/main.py. Accessed: 2019-09-24. 14

[35] Pytorch. Pytorch example on resnet. https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py. Accessed: 2019-10-15. 14

[36] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of European Conference on Computer Vision*, pages 525–542, 2016. 1, 2, 5, 7, 8

[37] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *Proceedings of International Conference on Learning Representations*, 2018. 3, 4

[38] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 5, 8

[39] Oran Shayer, Dan Levi, and Ethan Fetaya. Learning discrete weights using the local reparameterization trick. In *Proceedings of International Conference on Learning Representations*, 2018. 8

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of International Conference on Learning Representations*, 2015. 7, 11, 14

[41] Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. Tbn: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of European Conference on Computer Vision*, 2018. 1, 8

[42] Wikipedia. Multiplyaccumulate operation. https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation. Accessed: 2019-10-20. 12

[43] Chen Xu, Jianqiang Yao, Zhouchen Lin, Wenwu Ou, Yuanbin Cao, Zhirong Wang, and Hongbin Zha. Alternating multi-bit quantization for recurrent neural networks. In *Proceedings of International Conference on Learning Representations*, 2018. 1, 2, 3, 4, 6

[44] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of European Conference on Computer Vision*, pages 365–382, 2018. 1, 2, 4, 5, 6, 8

[45] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of European Conference on Computer Vision*, pages 184–199, 2018. 7

[46] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *Proceedings of International Conference on Learning Representations*, 2017. 2, 8

[47] Aojun Zhou, Anbang Yao, Kuan Wang, and Yurong Chen. Explicit loss-error-aware quantization for low-bit deep neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 9426–9435, 2018. 2

[48] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016. 2

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 l^{th} 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 $\boldsymbol{\alpha}_{l,g}$ w.r.t. 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: $\{\boldsymbol{w}_l\}_{l=1}^{L}$, $\{G_l\}_{l=1}^{L}$, $\mathrm{I}_{\max}$, $\sigma$
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$
for $l \leftarrow 1$ to $L$ do
    for $g \leftarrow 1$ to $G_l$ do
        Fetch and vectorize $\boldsymbol{w}_{l,g}$ from $\boldsymbol{w}_l$ ;
        Initialize $\boldsymbol{\epsilon} = \boldsymbol{w}_{l,g}$, $i = 0$ ;
        $\boldsymbol{B}_{l,g} = [\,]$ ;
        while $\|\boldsymbol{\epsilon} \oslash \boldsymbol{w}_{l,g}\|_2^2 > \sigma$ and $i < \mathrm{I}_{\max}$ do
            $i = i + 1$;
            $\boldsymbol{\beta}_i = \mathrm{sign}(\boldsymbol{\epsilon})$;
            $\boldsymbol{B}_{l,g} = [\boldsymbol{B}_{l,g}, \boldsymbol{\beta}_i]$;
            /* Find the optimal point
               spanned by $\boldsymbol{B}_{l,g}$                */
            $\boldsymbol{\alpha}_{l,g} = (\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{B}_{l,g})^{-1}\boldsymbol{B}_{l,g}^{\mathrm{T}}\boldsymbol{w}_{l,g}$ ;
            /* Update the residual
               reconstruction error            */
            $\boldsymbol{\epsilon} = \boldsymbol{w}_{l,g} - \boldsymbol{B}_{l,g}\boldsymbol{\alpha}_{l,g}$ ;
        $I_{l,g} = i$;
```

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 $\mathrm{sign}(\boldsymbol{\epsilon}) \bullet \boldsymbol{\epsilon} > 0, \forall \boldsymbol{\epsilon} \neq \boldsymbol{0}$, we have $\boldsymbol{\epsilon} \notin \mathrm{span}(\boldsymbol{B}_{l,g})$. 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. □

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.(25), where $\lambda \geq 0$.

$$\|\boldsymbol{\epsilon}\|_2^2 \leq \|\boldsymbol{w}_g\|_2^2(1 - \frac{1}{n-\lambda})^{I_g} \tag{25}$$

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,

$$r_s = \frac{N \times 32}{I \times N + I \times 32 \times \frac{N}{n}} \tag{26}$$

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 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 × 256 × 3 × 3 tensor) of AlexNet [20] (see Fig. 3). The full precision weights are provided by Pytorch [30].

We have found that there is not a significant difference between random groups or 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 to speed up the 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 ($\boldsymbol{w}_{c,:,h,w}$)

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

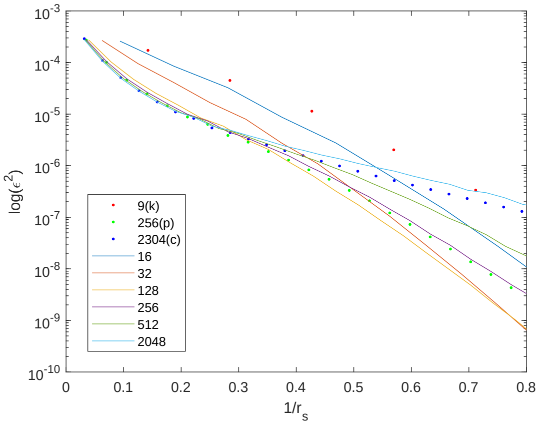


Figure 3. The curves about the logarithmic L2-norm of the average reconstruction error $\log(\|\boldsymbol{\epsilon}\|_2^2)$ related to the reciprocal of the storage compression rate $1/r_s$ (from the last convolution layer of AlexNet). The legend demonstrates the corresponding group sizes. 'k' stands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

are appropriate. Channel-wise $\boldsymbol{w}_{c,:}$ and sub-channel-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). In addition, considering that most of general microprocessors have already supported (floating-point) MAC unit [42], all above three inter-filter structures (quantized with the same average bitwidth) consume the same number of MAC operations when convoluted with (multiple) binary inputs.

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 l^{th} 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 do not always come from the same 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 l^{th} layer. There are n_l weights in each group, and G_l groups in the l^{th} 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 algorithm 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.*

$$M_0 = \sum_l \sum_g \text{card}(\boldsymbol{\alpha}_{l,g}) \tag{27}$$

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.(27) ;
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 $\ell^t$ ;
    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 learning rate in optimizing $\boldsymbol{B}_g$, and is decayed with 0.1 after half of epochs; the learning rate is set to 10^{-4} in optimizing $\boldsymbol{\alpha}_g$, and is also decayed with 0.1 after half of epochs.

B.2.1 Optimizing 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 process, 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: $Q$, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, 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 $\ell^q$ ;
    for $l \leftarrow L$ to 1 do
        Backward propagate gradient $\frac{\partial \ell^q}{\partial \hat{\boldsymbol{w}}_{l,g}^q}$ ;
        Update momentums of AMSGrad ;
        for $g \leftarrow 1$ to $G_l$ do
            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.(19)) ;
            Update $\boldsymbol{\alpha}_{l,g}^{q+1}$ 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, also 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). After initialization, we might need to retrain the model with several epochs of B.2 to recover the accuracy degradation caused by initialization.

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

```
Input: $P$, $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$, Training Data
Output: $\{\{\boldsymbol{\alpha}_{l,g}, \boldsymbol{B}_{l,g}, I_{l,g}\}_{g=1}^{G_l}\}_{l=1}^{L}$
for $p \leftarrow 1$ to $P$ do
    for $l \leftarrow 1$ to $L$ do
        Update $\hat{\boldsymbol{w}}_{l,g}^p = \boldsymbol{B}_{l,g} \boldsymbol{\alpha}_{l,g}^p$ ;
        Forward propagate convolution ;
    Compute the loss $\ell^p$ ;
    for $l \leftarrow L$ to 1 do
        Backward propagate gradient $\frac{\partial \ell^p}{\partial \hat{\boldsymbol{w}}_{l,g}^p}$ ;
        Compute $\frac{\partial \ell^p}{\partial \boldsymbol{\alpha}_{l,g}^p}$ with Eq.(14) ;
        Update momentums of AMSGrad in $\boldsymbol{\alpha}$
          domain ;
        for $g \leftarrow 1$ to $G_l$ do
            Update $\boldsymbol{\alpha}_{l,g}^{p+1}$ with Eq.(21) ;
```

Then, we need to determine the number of outer iterations R, *i.e.* how many times the pruning (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 r^{th} 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,

$$M^{r+1} = M^r \times (1 - 0.3) \qquad (28)$$

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 includes 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).

C. Implementation Details

C.1. LeNet5 on MNIST

The MNIST dataset 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

```
Algorithm 5: Adaptive Loss-aware Quantization
Input: Pretrained FP Weights $\{\boldsymbol{w}_l\}_{l=1}^L$, Structures $\{G_l\}_{l=1}^L$, $\mathrm{I}_{\max}$, $\sigma$, $T$, 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:                                        */
    Substitute $M^r$ into 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 ;
```

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 *kernelwise, kernelwise, subchannelwise(2), channelwise* respectively. The *subchannelwise(2)* structure means all input channels are sliced into two groups with the same size, *i.e.* the group size here is $\frac{800}{2} = 400$. After each pruning, the network is retrained to recover the accuracy degradation with 10 epochs of optimizing $\boldsymbol{B}_g$ and 20 epochs of optimizing $\boldsymbol{\alpha}_g$. The pruning ratio is 40%, and 6 times pruning (Step 1) are executed after initialization in the reported experiment (Table 2). After the last pruning, if the train loss is still not converged, we add another 30 epochs of optimizing steps (Sec. 3.3) with a gradually decayed learning rate (also applied in the following experiments of VGG and ResNet18/34).

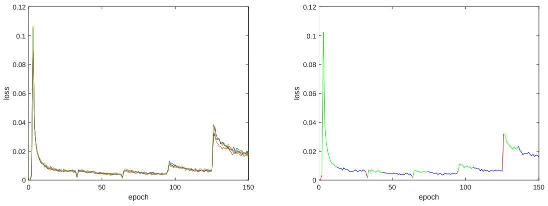
C.2. VGG on CIFAR10

The CIFAR-10 dataset 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 *channelwise, pixelwise, pixelwise, pixelwise, pixelwise, pixelwise, subchannelwise(16), subchannelwise(2), subchannelwise(2)* respectively. After each pruning, the network is retrained to recover the accuracy degradation with 10 epochs of optimizing $\boldsymbol{B}_g$ and 20 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), which is also the trial used to report results in Sec. 5.4.2.



(a) (b)

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.

C.3. ResNet18/34 on ILSVRC12

The ImageNet (ILSVRC12) dataset consists of 1.2 million high-resolution images for classifying in 1000 object classes. The validation set contains 50k images, which is 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].

The structures of each layer chosen for ALQ are all *pixelwise* except for the first layer (*channelwise*) and the last layer (*subchannelwise(2)*). After each pruning, the network is retrained to recover the accuracy degradation with 5 epochs of optimizing $\boldsymbol{B}_g$ and 10 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).

What is more, if we add another 50 epochs of our *STE with loss-aware* (see in Sec. 5.2) in the end, with a gradually decayed learning rate (10^{-4} decays with 0.95 per epoch), the final accuracy reported in Table 4 can be further boosted (see example results in the next paragraph). We think this is due to the fact that several layers have already been pruned to an average bitwidth below 1-bit (see Fig. 2). With such an extremely low bitwidth, maintained full precision parameters will help to calibrate some aggressive steps of quantization, which could slowly and fluctuatedly converge 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-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 (optimizing $\boldsymbol{B}_g$ with speedup in Sec. 3.3) can converge to a better local optimum, faster and more stable.

Following the process above, for example, ResNet18 with an average bitwidth of 2.0 (*i.e.* $2.0/32$ in Table 4) even yields a Top-1/5 accuracy of 70.0%/89.2%, which already exceeds its full precision version (69.8%/89.1%). In other words, ALQ can quantize ResNet18 with 2.0-bit (across all layers) *without any accuracy loss*. To the best of our knowledge, this is the first time that the quantized 2-bit ResNet18 can achieve the accuracy level of its full precision version, even if some prior schemes keep the first and last layers unquantized. ResNet18 with an average bitwidth of 1.0 (*i.e.* $1.0/32$ in Table 4) is also raised to a Top-1/5 accuracy of 67.8%/87.8%. These results further demonstrate the high-performance of the pipeline in ALQ.