MicroNet Challenge Submission - QualcommAI-MixNet

Team: Yash Bhalgat[1], Kambiz Azarian[1], Jinwon Lee[1], Jangho Kim[123], Hsin-Pai Cheng[124], Chirag Patel[1]
Contact: ybhalgat@qti.qualcomm.com

1 Submission description

In this submission, starting with MixNet-S [2], we use L2-norm based unstructured weight pruning to compress the network 1.9x. We use Learned Stepsize Quantization (LSQ) [1] to quantize the compressed model to a bit-width of 8 for both weights and activations of all layers. We use a Knowledge Distillation method based on Deep Mutual Learning [3] to boost the accuracy of our quantized model to the required 75% accuracy threshold.

The MicroNet score of our submission is **0.1819**. (Details on score calculation in Section 3)

2 Implementation Details

2.1 Quantization scheme

The quantization scheme [1] we use to quantize the weights and activations of our network is as follows:

$$\bar{v} = \lfloor clip(v/s, -Q_N, Q_P) \rceil \tag{1}$$

$$\hat{v} = \bar{v} \times s \tag{2}$$

Q_P and Q_N represent the number of positive and negative quantization levels. The $clip(z, -Q_N, Q_P)$ function returns z with values below $-Q_N$ set to $-Q_N$ and values above Q_P set to Q_P. This quantization-dequantizaiton scheme is implemented in `lsq_quantizer/utils/lsq_module.py`.

Weight quantization: We quantize all the weights in the Conv and Linear layers using symmetric quantization, i.e. $Q_N = 2^{b-1}$ and $Q_P = 2^{b-1} - 1$. The BatchNorm layers are NOT unquantized.

Activation quantization: All the activations which go into the Conv/Linear layers as inputs are quantized. Since the weights in the Conv/Linear layers are also quantized, all the operations inside these layers are b-bit/b-bit operations, where b is the bit-width for the layer. (*Quick note: since we use mixed-precision setting, the bit-width b for every layer is different*) For the activations, symmetric and asymmetric quantization are used interchangeably as follows:

1. The ReLU layers are simply replaced by asymmetric quantization layers with $Q_N = 0$ and $Q_P = 2^b$. As can be seen here, this setting of Q_N and Q_P automatically constrains the activations to be greater than 0 (in addition to quantizating them to the corresponding bit-width).

2. There is no ReLU before some of the Conv layers (e.g. `_conv_stem`) and the final `Linear` layer. Hence, we want to preserve both the +ve and -ve activations that go into these layers (unlike ReLU). So, we use symmetric quantization ($Q_N = 2^{b-1}$ and $Q_P = 2^{b-1} - 1$) as these input activations layers (namely, `*._in_act_quant`, `first_act`, `_head_act_quant0` and `_head_act_quant1`). For the details on these layers, refer `lsq_quantizer/utils/effnet.py`

[1] Qualcomm AI Research, an initiative of Qualcomm Technologies Inc. [2] This work was done when the author was an intern at Qualcomm AI Research [3] MIPA Lab, Seoul National University [4] CEI Lab, Duke University

3. **Swish activation:** Given the shape of the swish function, we also have to allow negative activations, but not symetrically. Hence, in our implementation, we replace the Swish layers by activation quantization layers with $Q_N = 1$ and $Q_P = 2^b - 1$, so the rounded integer values range from $\{-1$ to $2^b - 1\}$ providing just enough room for the -ve activations.

Please refer to the `get_constraint` function in `lsq_quantizer/utils/utilities.py` for the setting of Q_N and Q_P levels for different layers.

2.2 Training

The parameter s is trainable. The gradient update of the parameter s is as follows:

$$\frac{\partial \hat{v}}{\partial s} = \begin{cases} -v/s + \lfloor v/s \rceil & \text{if } |v/s| < L \\ \hat{v}/s & \text{otherwise} \end{cases}$$

For each layer in the network, there is one s-parameter for weights and one s-parameter for activations.

For training, we have 3 different learning rates (similar to [1]):

1. `learning_rate`: The usual learning rate for the weights of the network
2. `weight_lr_factor`: We need a different learning rate for the s parameter for the weights. We define this learning rate as $weight\_lr\_factor \times learning\_rate$
3. `act_lr_factor`: This is same as above, just for the s parameter for the activations.

2.3 Deep Mutual Learning

We apply the knowledge distillation method to boost quantized network accuracy. We jointly train the teacher network (Full-precision) and the student network (Quantized network) simultaneously using Kullback–Leibler divergence (KL) similar to DML [3]. Firstly, we calculate the posterior of the teacher network and the student network with as below equation:

$$p_i(\mathbf{z}_k; T) = \frac{e^{z_k^i/T}}{\sum_j^m e^{z_k^j/T}} \tag{3}$$

$\mathbf{z}_k$ refers to logit of k network, which means student or teacher ($k = \{s, t\}$). i means the class. T means the temperature value to make distribution softer. Then, we update each networks with cross entropy and KL loss as below:

$$studentloss = \mathcal{L}_{ce}^s + \mathcal{L}_{KL}(z_t || z_s) \tag{4}$$

$$teacherloss = \mathcal{L}_{ce}^t + \mathcal{L}_{KL}(z_s || z_t) \tag{5}$$

$\mathcal{L}_{ce}$ and $\mathcal{L}_{KL}$ refer to cross entropy and KL loss, respectively.

3 Parameter and MAC counting

The script `lsq_quantizer/utils/micronet_score.py` implements the score counting function. The score counting code has been derived this repo: flops-counter.pytorch.

- Bitmask overhead is considered while calculating the parameters for sparse Conv/Linear layers.
- According to the rules of the competition, additions and multiplications are calculated separately.

- As mentioned before, both weights and input activations for Conv and Linear layers are quantized. In all our submissions, we quantize the weights and activations of a particular layer to the same bit-width (WbAb) to keep the flops counting simple. No quantization is considered for the BatchNorm.

Refer `lsq_quantizer/flops_counter.py`:

- `compute_average_flops_cost` does the MAC counting (here, MAC = multiply-add count)
- `get_model_parameters_number` does the parameter counting

Verfication: To verify the counting score is correct, we input the baseline model for CIFAR100, i.e. WRN-28-10 to check if we can get the same numbers as mentioned here. The numbers we get are:

- #params = 37678095, MAC = 10491105563
- The numbers mentioned on the scoring and submission page are: 36.5M params and 10.49B MAC
- The slight discrepancy in the #params counting is due to the bitmask overhead calculation

Please email the authors for any questions about the implementation.

4 Results

We start from a pretrained checkpoint for MixNet-S and compress it 2.22x using L2-norm based unstructured pruning. On quantizing this compressed model to W8A8. We use Deep Mutual Learning (DML) with full-precision MixNet-M as the teacher and our compressed+quantized model as the student. With DML, we were able to recover the accuracy to 75.02%.

Model	**Accuracy**	**#params**	x+ **count**	**score**
MixNet-S (Full precision)	75.77%	4.1M	0.49G	1.04
+ 1.9x compression	75.24%	2.2637M	0.3827G	0.6551
+ Quantize with DML (W8A8)	**75.02%**	**0.6465M**	**0.1032G**	**0.1819**

Table 1: This table shows the improvement in MicroNet score with every step. The last row is our submission.

5 Reproducibility

5.1 Training

The main file which calls the LSQ and DML methods is `lsq_quantizer/lsq_main_KD.py`. All the arguments to this script are defined in `lsq_quantizer/utils/lsq_train.py`.

The quantization procedure starts from a pretrained full-precision model. The pretrained checkpoint should be placed in the `<model_root>` location with the name `<model-name>.pth`.

In our submission, we quantize all layers, including the first and last layers. This is enabled by the `quan_first` and `quan_last` options.

In this submission, we quantize a compressed model. To ensure that the zero weights don't update during quantization-aware fine-tuning, the gradients for these weights should also be zero. This is enabled by the `--pruned` option.

The following command should be used for the quantization-aware training:

```
python lsq_quantizer/lsq_main_KD.py \
        --dataset imagenet --data_root <imagenet_path> \
        --weight_bits 8 --activation_bits 8 \
        --prefix 1.9x_W8A8_baselr0.001_lrW0.005_lrA1_exp_ \
        --learning_rate 0.001 --weight_lr_factor 0.005 --act_lr_factor 1 \
        --lr_scheduler exp --total_epoch 100 --batch_size 120 \
        --network mixnet_s --model_name mixnet_s \
        --model_root <model_root> \
        --quan_first --quan_last --pruned
```

5.2 Checkpoint evaluation

Use the following command to evaluate the checkpoint:

```
python3 lsq_quantizer/evaluation.py \
        --model_path final_checkpoint/mixnet_s.pth \
        --data_root <imagenet_path> \
        --weight_bits 8 --activation_bits 8
```

References

[1] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.

[2] Mingxing Tan and Quoc V Le. Mixnet: Mixed depthwise convolutional kernels. *arXiv preprint arXiv:1907.09595*, 2019.

[3] Ying Zhang, Tao Xiang, Timothy M Hospedales, and Huchuan Lu. Deep mutual learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018.