MicroNet Challenge Submission - QualcommAI-M0

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

1 Submission description

Starting with the EfficientNet-B0 [2] architecture, we use Learned Stepsize Quantization (LSQ) [1] to quantize the model to a bit-width of 6 for both weights and activations of all layers. We use a Knowledge Distillation method based on Deep Mutual Learning [3] to increase the accuracy of our quantized model to the required 75% accuracy threshold. The MicroNet score of our submission is **0.22**.

2 Implementation Details

2.1 Quantization

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

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

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

This quantization-dequantizaiton scheme is implemented in `lsq_quantizer/utils/lsq_module.py`.

Symmetric and asymmetric quantization: In our submission, we use 6-bits. Hence, if we use symmetric quantization for a layer, the quantity $\bar{v}$ are constrained to the range:

$$[-31, -30, -29, \ldots, -1, 0, 1, \ldots, 29, 30, 31]$$

And if we use asymmetric quantization for a particular layer, the quantity is constrained to the range:

$$[0, 1, 2 \ldots, 61, 62, 63]$$

This is implemented in the `get_constraint` function in `lsq_quantizer/utils/utilities.py`

Weight quantization: We quantize all the weights in the Conv and Linear layers to 6-bits. The BatchNorm layers are kept unquantized. For weights, symmetric quantization is used.

Activation quantization: All the activations which go into the Conv/Linear layers as inputs are quantized to 6-bits. Hence, all the operations inside the Conv/Linear layers are 6-bit/6-bit operations. For the activations, symmetric and asymmetric quantization are used interchangeable as follows:

1. The ReLU layers are simply replaced with the asymmetric quantization layers. Because, as can be seen above, the asymmetric quantization layer automatically constrains the activations to be greater than 0 (in addition to quantizating them to the corresponding range).
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. So, we use symmetric quantization for these input activations (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

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 learning rates as described in [1]. We modify it and we have 3 learning rate parameters as follows:

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 2-step Knowledge Distillation

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. As mentioned before, both weights and input activations for Conv and Linear layers are quantized. No quantization is considered for the BatchNorm. Refer `lsq_quantizer/flops_counter.py`:

- `compute_average_flops_cost` does the MAC counting
- `get_model_parameters_number` does the parameter counting

Please email the authors for any questions about the implementation.

4 Results

We start from a pretrained checkpoint for EfficientNet-b0 and quantize it using LSQ. On quantizing it to W6A6, we observe a drop in performance by 1.9%. We use Deep Mutual Learning (DML) with full-precision EfficientNet-b1 as the teacher and our W6A6 model as the student. With DML, we were able to recover the accuracy to 75.1%.

Model	**Accuracy**	**#params**	**MAC**	**score**
EfficientNet-b0	76.10%	5.3M	0.39G	1.1
+ LSQ (W6A6)	74.21%	1025740.75	91520157.25	0.2268
+ DML	**75.10%**	1025740.75	91520157.25	**0.2268**

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 `efficientnet-b0.pth` (for quantizing EfficientNet-b0, of course).

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

```
python lsq_quantizer/lsq_main_KD.py \
        --dataset imagenet --data_root <imagenet_path> \
        --weight_bits 6 --activation_bits 6 \
        --prefix W6A6_KD_effb1_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 140 \
        --network efficientnet-b0 --model_name efficientnet-b0 \
        --model_root <model_root> \
        --quan_first --quan_last
```

5.2 Checkpoint evaluation

Use the following command to evaluate the checkpoint:

```
python3 lsq_quantizer/evaluation.py \
        --model_path final_checkpoint/efficientnet-b0.pth \
        --data_root <imagenet_path> \
        --weight_bits 6 --activation_bits 6
```

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. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 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.