MicroNet Challenge Submission - QualcommAI-nanoWRN

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, we introduce a novel evolutionary search-and-eval algorithm to search for the layer-wise bit-widths (mixed-precision configuration for weights and activations) that give the best trade-off between model performance and model efficiency. Our search-and-eval method is built upon LSQ (Learned Step Size Quantization) [1]. More details about this in Section 3.

Starting with WideResNet-40-4 (depth=40, widen_factor=4) [4], we use L2-norm based unstructured pruning to compress the network by a factor of 2.22x. We then use our search-and-eval method to find the layer-wise bit-widths for our compressed network. We use a Knowledge Distillation method based on Deep Mutual Learning [5] to boost the accuracy of our quantized model to the required 80% accuracy threshold.

The MicroNet score of our submission is **0.0720**. (Details on score calculation in Section 4)

2 Implementation Details

Before going into the details of our search-and-eval method, we will describe the underlying vanilla quantization scheme and the training methodology. In Section 3, we will go into the details of the search-and-eval algorithm for mixed precision setting.

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] 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

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`

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. (Note that: Although WRN doesn't use any swish, this technique was useful for our other submissions to ImageNet track.)

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 [5]. 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 Layer-wise bit-width search using an evolutionary algorithm

Every layer in a deep network has different sensitivity to quantization [3]. Accordingly, it gives a chance to quantize further by adaptively changing bit-width per layer. The key challenge here is how to quickly optimize bit-width for each layer while minimizing accuracy drop. In this submission, we first leverage an evolutionary algorithm called CEM [2] for bit-width search. We first combine a micronet score and accuracy into a single reward function and then let CEM to optimize it by evolving the configuration of bit-width per layer. During this combinatorial optimization, we estimate the accuracy with a bit-skipping method without any fine-tuning, which reduces search time significantly. Finally, we finetune the model once the best bit-width configuration has been found by CEM.

3.1 Search:

- In our method, for every layer, the bit-width of input activations = bit-width of layer weights
- Each layer can have a bit-width in the range $\{2, 3, ..., b-1, b\}$. Hence, our search space consists of $\{$WbAb, W$(b-1)$A$(b-1)$, ..., W2A2$\}$ options for each layer. So, if we have N layers, the search space has $(b-1)^N$ possible bit-width configurations for the network.
- We use the Cross Entropy Model [2] method to search in this space of configurations. But the CEM method needs a reward value for each configuration. Our reward function is a combination of (1) pre-finetuning accuracy of the configuration (described in detail next) and (2) the MicroNet score of that mixed-precision configuration.

3.2 Eval:

We estimate the post-finetuning performance of each bit-width configuration using the pre-finetuning accuracy. We calculate the pre-finetuning accuracy as follows:

1. Since the maximum bit-width allowed is b, we first train our model with WbAb setting for all layers.
2. For a given layer-wise bit-width configuration, we perform a forward pass as follows - if a layer has a bitwidth $t \leq b$, we only use 2^t of the 2^b quantization levels. For example, if $b = 6$ and $t = 5$, we quantize the weights and input activations of that layer to the levels $[0, 2, 4, \dots, 58, 60, 62]$. Notice that we skip alternate quantization levels in this case, hence using only 2^5 levels. Since the scale parameters are tuned for the range $\{0, 63\}$, skipping quantization levels in this manner gives a good estimate of the accuracy of a given model when that layer would be quantized to 5 bits.

3.3 Implementation

The *eval* method as described gives us an estimate of the pre-finetuning accuracy of any given layer-wise bit-width configuration. We calculate the MicroNet score of any configuration using the code described in Section 4. Our reward function favours a high pre-finetuning accuracy and a low MicroNet score. The exact steps in this submission are as follows:

1. Compress full precision WideResNet-40-4 2.22x using L2-norm based untructured pruning
2. Quantize this compressed network to W4A4 (except first and last layer)
3. Use CEM search-and-eval method to search for the layer-wise bit-width configuration. The maximum bit-width allowed per layer in the search method is $b = 4$.
4. With the searched mixed-precision bit-widths configuration, fine-tune the network using Deep Mutual Learning method described in Section 2.3 with WideResNet-40-6 as teacher network.

In this submission, we don't quantize the first Conv layer and the last FC layer. So, the model has 39 quantizable layers. The final bit-width configuration we use is:

```
[4, 4, 4, 4, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 4, 2, 4]
```

The implementation of the CEM search-and-eval method can be found in `lsq_quantizer/cem_eval.py`.

4 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.

5 Results

We train WideResNet-40-4 and WideResNet-40-6 from scratch (we use this full precision WRN-40-6 as the teacher network in all our steps). We compress the trained WRN-40-4 2.22x using L2-norm based pruning. Then we quantize this model to W4A4 and use our evolutionary algorithm to find a set of layer-wise bit-widths. Lastly, to finetune the quantized network with this bit-width configuration. With DML, we were able to recover the accuracy to 80.02%.

Model	Accuracy	#params	x+ count	score
WideResNet-40-4 (Full precision)	79.08%	8.9M	2.6G	0.4917
+ 2.22x compression	79.10%	4.1120M	1.3782G	0.2440
+ Quantize (W4A4)	80.05%	0.6541M	0.7012G	0.0848
+ CEM search-and-eval	**80.02%**	**0.5470M**	**0.5987G**	**0.0720**

Table 1: This table shows the improvement in MicroNet score with every step. Every step was fine-tuned using DML. The last row is our submission.

6 Reproducibility

6.1 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.

```
python lsq_quantizer/lsq_main_KD.py \
        --dataset cifar100 \
        --weight_bits 4 --activation_bits 4 \
        --prefix 2.22x_W7A7_GREEDY_baselr0.001_lrW0.005_lrA1_exp_ \
        --learning_rate 0.001 --weight_lr_factor 0.01 --act_lr_factor 1 \
        --lr_scheduler exp --total_epoch 100 --batch_size 800 \
        --network wrn40_4 --model_name wrn40_4 \
        --model_root <model_root> \
        --pruned --greedy
```

6.2 Checkpoint evaluation

Use the following command to evaluate the checkpoint:

```
python3 lsq_quantizer/evaluation.py \
        --model_path final_checkpoint/wrn40_4.pth \
        --weight_bits 4 --activation_bits 4 --cem
```

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] István Szita and András Lörincz. Learning tetris using the noisy cross-entropy method. *Neural computation*, 18(12):2936–2941, 2006.

[3] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[4] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[5] 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.