Memory Efficient Training using Lookup-Table-based Quantization for Neural Network

Kazuki Onishi*, Jaehoon Yu†, and Masanori Hashimoto*
* Osaka University, Osaka, Japan
Email: {k-onishi, hasimoto}@ist.osaka-u.ac.jp
† Tokyo Institute of Technology, Tokyo, Japan
Email: yu.jaehoon@artic.iir.titech.ac.jp

***Abstract*—Modern neural networks require a tremendous number of parameters, which causes unaffordable requirements of memory and computation resources for embedded systems. To tackle this issue, we propose a LUT-based training method in this paper. The proposed method consists of two components: cluster swap and factorization. Cluster swap is an extension to the quantization process in Deep Compression that overcomes its drawback of unstable training by reassigning each parameter to the proximate cluster. Factorization is a computation trick to reduce the computational cost of neural networks. The experimental results show that the proposed method can decrease memory usage for forward and backward processes to 22.2% and 60.0% ,respectively, and reduce the number of multiplications to 11.7%, with 1.41% accuracy loss.**

***Index Terms*—neural network, lookup table training, cluster swap, Deep Compression, hardware acceleration**

I. Introduction

The demand for mobile applications using neural networks is increasing. These neural networks must be trained in advance to a sufficient level of accuracy. Neural networks are usually trained on powerful computers, since they require a large amount of hardware resources corresponding to the size of the network. On the other hand, embedded systems capturing and processing data in real time need to perform training with limited computational resources. In that case, training becomes a challenging issue due to increasing hardware resources which is brought by a tendency to grow the size of the network. In 1998, LeNet-5 [1] used only 40K parameters, but Inception-v4 [2] presented in 2016 has 100M parameters.

A larger number of parameters increase not only computation cost, but also memory requirement, both of which highly affect the amount of energy consumption. TABLE I shows that a DRAM fabricated in 45nm process requires 640pJ per operation, which is up to 711× more energy consumption than other operations. Due to the limitation of memory resources in embedded systems, it is difficult to store all parameters of a state-of-the-art neural network in on-chip memory. Therefore, embedded systems load parameters from off-chip DRAMs, increasing energy consumption as a result. Additionally, the growth in computational cost resulting from the increase in parameters is an obstacle to the practical use of neural networks in embedded systems.

TABLE I
Energy Consumption for 45nm CMOS process [4].

Operation	Energy[pJ]	Relative Cost
32 bit DRAM Access	640	711
32 bit SRAM Access	5	5.6
32 bit float MULT	3.7	4.1
32 bit float ADD	0.9	1

These problems are more conspicuous in the training phase, as shown in Fig. 1. Comparing memory requirements between conventional training and inference phases, ((a) and (c) in Fig. 1, respectively), the training phase requires additional memory for computation history and gradient calculation, which are used for differentiating output activations and updating weights.

A well-known solution to memory requirements is quantization. Quantization is a method for representing a broad set of values by a small set of values. For reducing neural network memory and computation resources, Deep Compression [3] uses quantization adopting lookup tables (LUTs), which substitutes words of LUTs for network parameters. In [3], Song *et al.* show that this LUT-based quantization can successfully reduce the size of state-of-the-art neural networks for the inference phase as shown in (a) and (b) of Fig. 1. The key to the success of Deep Compression is the repetitive retraining process after quantization, which recovers from the accuracy drop caused by quantization. Instead, if we can apply this retraining in the training phase with randomly initialized parameters, we can also reduce the memory requirements for the training phase, as shown in (d) of Fig. 1. As a result, it would be possible to realize the whole LUT-based training process and implement an embedded system that unifies both LUT-based training and inference.

In this paper, we analyze Deep Compression, describe challenging problems for using it in initial training, and propose a LUT-based training method. The proposed method is evaluated with LeNet-5 and the MNIST dataset [5]. To the best of our knowledge, this paper is the first attempt to train LUT-based

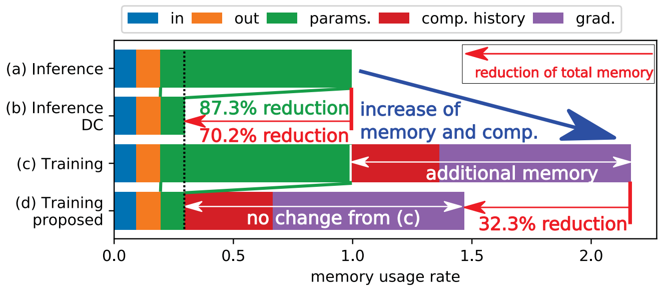
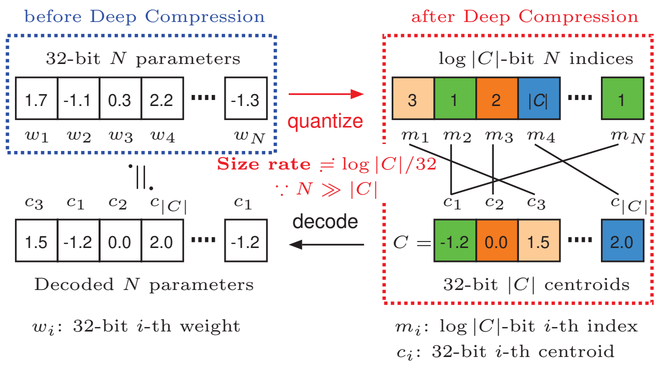


Fig. 1. Memory requirement for training and inference in CONV2 of LeNet-5.



before Deep Compression
after Deep Compression
32-bit N parameters
1.7
-1.1
0.3
2.2
-1.3
w_1 w_2 w_3 w_4 w_N
quantize
$\log|C|$-bit N indices
3
1
2
$|C|$
1
m_1 m_2 m_3 m_4 m_N
Size rate $= \log|C|/32$
$\because N \gg |C|$
c_3 c_1 c_2 $c_{|C|}$ c_1
1.5
-1.2
0.0
2.0
-1.2
decode
c_1 c_2 c_3 $c_{|C|}$
$C =$
-1.2
0.0
1.5
2.0
Decoded N parameters
32-bit $|C|$ centroids
w_i: 32-bit i-th weight
m_i: $\log|C|$-bit i-th index
c_i: 32-bit i-th centroid

Fig. 2. Underlying idea of Deep Compression.

quantized network from scratch without full-precision weights.

II. Conventional Quantization

In this section, we briefly explain Deep Compression, in which the proposed method is based on, and describe the problem in the training phase to be addressed. The remarkable inference performance of recent neural networks comes at the cost of billions of parameters, which entail high requirements of memory and computation resources. pproach to this issue is quantization. The numeric precision of parameters represents the resolution of non-linearly transformed spaces of neural networks. Generally, quantization approaches alleviate memory and computation requirements by representing the space with a small set of numbers. So far, existing quantization methods have achieved a great success in the inference phase of state-of-the-art neural networks.

The most effective quantization methods are using binary [6], [7], ternary [8], [9], or a bigger set of non-linearly spaced values. Compared with networks using 32-bit floating point, binary and ternary quantization can achieve $32\times$ and $16\times$ memory savings, respectively. This feature brings a huge advantage in reducing memory and computation requirements. However, excessive quantization always degrades inference performance, and this limitation in binary and ternary is becoming increasingly severe in newer models. On the other hand, non-linear quantization provides us with a satisfying solution to this problem.

Deep Compression [3] is a well-known non-linear quantization method that uses LUTs and addresses instead of parameters, as shown in Fig. 2. Even only with such simple replacement, this method can reduce the required memory size to less than one fifth. Deep Compression consists of three processes: pruning, quantization, and encoding, but we focus only on quantization in this paper. The main components of quantization in Deep Compression are clustering and retraining. In the clustering process, Deep Compression applies K-means clustering to network parameters, obtaining a centroid from each cluster. Since the approximation error of each centroid can cause an accuracy drop, Deep Compression finely tunes the centroids by retraining and updating each centroid with the aggregate sum of back-propagation gradients of the parameters in the corresponding cluster. This update process can be described by the following equation:

$$c_j^{(i+1)} = c_j^{(i)} - \rho \sum_{k \in \mathbb{I}_j} \frac{\partial L}{\partial w_k^{(i)}}, \tag{1}$$

where $c_j^{(i)}$ and $w_k^{(i)}$ are, respectively, the j-th centroid and the k-th weight after i retraining iteration, ρ is the learning rate, L is the loss function, and $\mathbb{I}_j$ is the set of weight indices assigned to the j-th cluster. Fig. 3a illustrates how Deep Compression clusters parameters and how it updates centroids after clustering, where circles and squares represent parameters and centroids, respectively, and each color represents each cluster. The crucial point is that the cluster assignment hardly changes during the centroid update, since parameters are already close to the local minimum after training, and therefore, the gradients are not large enough to force them to another cluster.

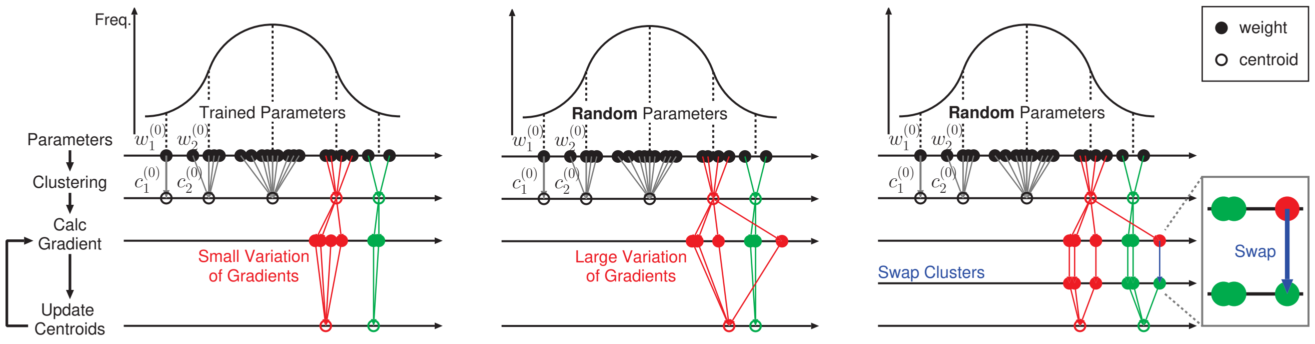
This retraining process provides us both inspiration and a challenging problem. If we could use LUT-based quantization from the beginning of the training phase, we could save memory and computation resources since we would not need to store all intermediate data to calculate gradients for back-propagation. As shown in Fig. 3b, in contrast to the update of retraining for fine-tuning in Fig. 3a, the initial training uses randomized parameters, which have large gradients toward local minima. As a result, each cluster shows a significant deviation, and the training becomes unstable.

III. Proposed LUT-based Training Method

This section proposes a method for memory saving and computation reduction. The proposed method consists of two components: cluster swap and factorization. Cluster swap solves the instability problem mentioned above and enables LUT-based training from randomly initialized parameters. As a result, it saves a large amount of memory resources. Factorization, on the other hand, exploits a computational advantage inherent in LUT-based quantization methods and reduces the number of multiplications used in both training and inference phases. The following subsections explain their details.

A. LUT-based training with cluster swap

As mentioned in Sect. II, the problem of LUT-based training is the initial cluster assignment is far apart from the optimal one. To solve this problem, the proposed method uses cluster swap to reassign each parameter to the cluster with the nearest centroid after the gradient calculation, as shown in Fig. 3c. By



(a) Retraining of Deep Compression (b) Initial training using Deep Compression (c) Initial training using the proposed method

Fig. 3. Comparison of LUT-based training processes.

adjusting each parameter assignment, this method alleviates the instability in training.

Cluster swap requires additional computation for reassigning parameters and then recalculating gradients. To suppress this increase in computational cost, we use a mathematically identical but more efficient calculation process. Without LUTs, parameters are updated as follows:

$$w_j^{(i+1)} = w_j^{(i)} - \rho \frac{\partial L}{\partial w_j^{(i)}}. \tag{2}$$

Then, cluster swap selects the nearest cluster based on

$$m^* = \arg\min_m \|w_j^{(i+1)} - c_m^{(i)}\|, \tag{3}$$

where m^* is the nearest cluster index of the j-th parameter after i-th cluster swap. Centroid update recalculates the gradient of each parameter reassigned to another cluster in the swap process because the previous gradient is calculated from the previous cluster, not from the current cluster. To avoid the recalculation from scratch, we assume that the value of the updated parameter does not change even if it belonged to the reassigned cluster from the previous update:

$$w_j^{(i+1)} = w_j^{(i)} - \rho \frac{\partial L}{\partial w_j^{(i)}} = w_j'^{(i)} - \rho \frac{\partial L}{\partial w_j'^{(i)}}, \tag{4}$$

where $w_j'^{(i)}$ is the reassigned weight, which is equal to the nearest cluster centroid c_{m^*}. Based on this assumption, gradient after reassignment can be calculated by

$$\frac{\partial L}{\partial w_j'^{(i)}} = \frac{\partial L}{\partial w_j^{(i)}} - \frac{w_j^{(i)} - w_j'^{(i)}}{\rho}. \tag{5}$$

With this gradient for the reassigned cluster, we can update centroids as follows:

$$c_j^{(i+1)} = c_j^{(i)} - \rho \frac{1}{|\mathbb{I}_j|} \sum_{k \in \mathbb{I}_j} \frac{\partial L}{\partial w_k'^{(i)}}. \tag{6}$$

This calculation is almost the same as in Deep Compression [3], but we use a value much smaller than the total of gradients, since we take the average to avoid unstable fluctuation of update in the initial training phase. Also, the proposed method only adds the calculation described of (3) and (5), which is negligible compared with the entire computation.

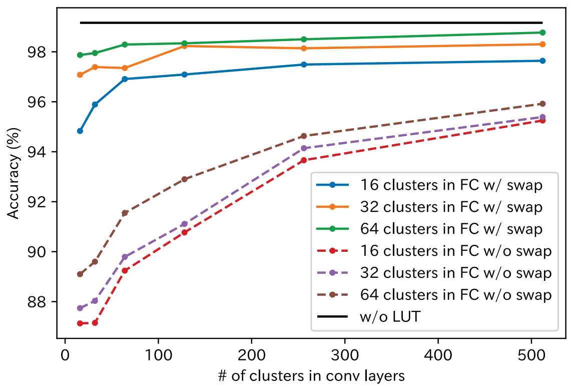


Fig. 4. Comparison of accuracy changes with increasing clusters.

B. Reducing multiplications with factorization

LUT-based quantization uses a small number of centroids instead of billions of parameters. This means a few multipliers are repeatedly multiplied to various multiplicands. Therefore, it is possible to factorize multiplications using an identical multiplier. We integrate this factorization into the calculation of each layer's output, which results in a reduction in the number of multiplications in both the training and inference phases. Generally, the output y of MAC operations can be described as

$$y = \sum_i w_i x_i, \tag{7}$$

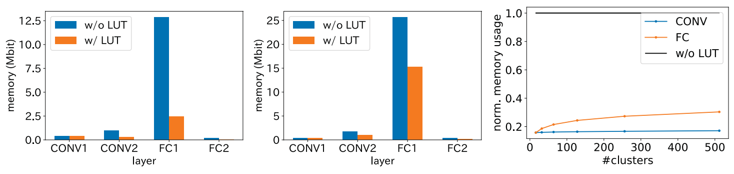
where w_i and x_i are the i-th parameter and input, respectively. In quantized networks, however, we can factorize multiplications and operate the same calculation as follows:

$$y = \sum_j c_j \sum_{k \in \mathbb{I}_j} x_k. \tag{8}$$

As a result, factorization can reduce the number of multiplications to $|C| \,/\, |K|$ compared with the original calculation, where $|C|$ and $|K|$ are the number of clusters and the dimensionality of kernels, respectively.

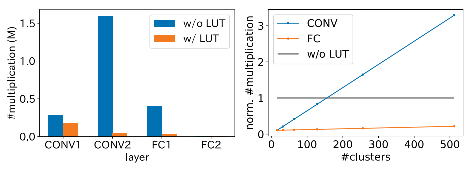
IV. Evaluation

We implemented the proposed method with PyTorch and evaluated it with LeNet-5 [1] on the MNIST dataset. Since



(a) Forward (b) Backward (c) Memory usage changes with increasing clusters

Fig. 5. Analyses of memory usage.



(a) Comparison of #MULT operations with and without LUT (b) Computational cost changes with increasing clusters

Fig. 6. Computaion reduction of the proposed method.

TABLE II
COMPARISON OF MEMORAY USAGE AND CALCULATION COST AT 16 CLUSTERS FOR CONV AND 64 CLUSTERS FOR FC.

Model	Accuracy (diff.) [%]	Memory usage [bit]		# MULT operations
		Forward	Backward	
w/o LUT	99.28 (0.00)	14.4M	28.1M	2.29M
w/ Swap	97.87 (-1.41)	3.21M	16.9M	268K
w/o Swap	89.10 (-10.18)	3.21M	16.9M	268K

LeNet-5 has two types of trainable layers, convolution (CONV) layers and fully connected (FC) layers, we separately adjusted the number of clusters for each type. Also, in our implementation, each layer has an independent LUT for storing and updating its centroids. This section provides analyses of the proposed LUT-based training method in terms of inference accuracy, memory saving, and computation reduction.

Fig. 4 shows the comparison of inference accuracy between LUT-based training processes with and without the cluster swap, where solid colored lines and dashed lines represent the accuracy changes with and without the cluster swap, respectively. The black line is the upper bound obtained from the original LeNet-5. As Fig. 4 shows, more clusters enable better accuracy for both layer types regardless of training processes, which is intuitively reasonable. However, Fig. 4 also shows that the proposed method using cluster swap largely outperformed competitors especially when using a small number of clusters. Note that the solid green line achieved about 98% accuracy at 16 clusters for CONV and 64 clusters for FC, which is approximately 9% better accuracy compared with its counterpart and only 1.41% accuracy drop from the original.

TABLE II provides the result of memory usage and computation cost at 16 clusters for CONV layers and 64 clusters for FC layers, where the models, ”w/o LUT”, ”w/ Swap”, and ”w/o Swap”, represent the conventional training with 32-bit floating points, the proposed LUT-based training, and the LUT-based training without swap, respectively. As shown in TABLE II, the proposed method with swap outperformed its competitor although both LUT-based training methods reduced memory usage to 22.2% in the forward process and 60.0% in the backward process while they reduced computation cost to 11.7%, where the forward memory usage includes the size of inputs, outputs, and parameters, and the backward memory usage additionally includes the gradients of parameters.

For more details, Figs. 5a and 5b show the breakdown of forward and backward memory usage. Also, Fig. 5c plots the memory usage changes for CONV and FC layers with increasing clusters. layer FC1 occupies the majority of memory usage in LeNet-5 for forward and backwards processes, and the proposed method works successfully for both cases. Also, Fig. 5c indicates that both normalized memory usages show logarithmic growth against the number of clusters, which is a desirable feature since we can thus apply the proposed method to larger neural networks with small costs.

Also, for the analysis of computation reduction, Fig. 6a provides the breakdown of the number of MULT operations, and Fig. 6b shows the normalized computational cost changes with increasing clusters. As shown in Fig. 6a, the proposed method can reduce a large amount of calculation. Although the computation cost is linearly proportional to the number of clusters in Fig. 6b, it hardly becomes a problem because we already know that a small number of clusters is enough for training.

V. CONCLUSION

In this paper, we proposed the LUT-based training method based on Deep Compression using cluster swap and factorization. The underlying idea is simple: cluster swap reassigns parameters with large gradients to the nearest cluster in the quantization of Deep Compression, and factorization reduces multiplication by summing up multiplicands in advance. For evaluation, we implemented the proposed method with PyTorch and tested it with LeNet-5 on the MNIST dataset. Experimental results showed that this simple approach can reduce forward and backward memory usage, respectively, to 22.2% and 60.0% with a 1.41% accuracy drop. Also, it can reduce the number of multiplications to 11.7%. Although the model used for this evaluation is simple, the proposed method worked successfully without any prior model-specific optimization, suggesting that it will also succeed when applied to other networks. In future work, we will verify its performance with more sophisticated neural networks such as Inception-v4 [2].

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP19H04079

REFERENCES

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[2] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
[3] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proceedings of International Conference on Learning Representations*, 2016.
[4] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.
[5] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[6] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of Advances in Neural Information Processing Systems*, 2016, pp. 4107–4115.
[7] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proceedings of European Conference on Computer Vision*. Springer, 2016, pp. 525–542.
[8] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.
[9] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *Proceedings of International Conference on Learning Representations*, 2016.