

# Structured Sparse Ternary Weight Coding of Deep Neural Networks for Efficient Hardware Implementations

Yoonho Boo and Wonyong Sung

Department of Electrical Engineering and Computer Science

Seoul National University

Seoul 151-744, South Korea

Email: yhboo.research@gmail.com; wysung@snu.ac.kr

**Abstract**—Deep neural networks (DNNs) usually demand a large amount of operations for real-time inference. Especially, fully-connected layers contain a large number of weights, thus they usually need many off-chip memory accesses for inference. We propose a weight compression method for deep neural networks, which allows values of +1 or -1 only at predetermined positions of the weights so that decoding using a table can be conducted easily. For example, the structured sparse (8,2) coding allows at most two non-zero values among eight weights. This method not only enables multiplication-free DNN implementations but also compresses the weight storage by up to x32 compared to floating-point networks. Weight distribution normalization and gradual pruning techniques are applied to mitigate the performance degradation. The experiments are conducted with fully-connected deep neural networks and convolutional neural networks.

**Index Terms**—Deep neural networks, weight storage compression, structured sparsity, fixed-point quantization, network pruning.

## I. INTRODUCTION

Deep neural networks (DNNs) show high performance in various classification problems. However, the implementation of them requires a much increased number of arithmetic operations when compared to traditional pattern recognition and classification algorithms [1], [2]. In particular, fully-connected layers of fully-connected deep neural networks (FCDNNs) or convolutional neural networks (CNN) contain a large number of parameters, which makes it difficult to implement them using resource-limited hardware. The number of weights usually exceeds millions, which incurs many off-chip memory accesses and large power consumption. When all the weights are expressed as ternary values (+1, 0, and -1), it is possible not only to reduce the off-chip memory access but also to remove multiplications.

Weight quantization is a straightforward way of reducing the size of parameters [3]–[9]. Many DNNs show high resiliency on weight quantization [10]. In particular, the precision of the weight can be lowered to 2-bit (+1, 0 and -1) without much performance degradation by retraining the quantized networks [3], [6], [7]. This ternary representation can compress the weight by x16 when compared to the 32-bit floating-point network. In recent years, there were some trials to

represent the DNN weight using the 1-bit binary (+1 and -1) format to increase the compression ratio (x32) and perform inference using only logical operations [8]. However, it still shows considerable performance degradation. Compression of DNNs and CNNs employing a few data compression techniques has been developed in [9]. This method prunes small valued weights to remove 89% and 92.5% of connections for AlexNet [11] and VGG-16 [12] and applies vector quantization. Also the final weights are compressed using a compressed sparse row/compressed sparse column (CSC/CSR) format with relative index and Huffman coding to achieve x35, x49 times weight storage compression. However, not only the decoding method used in this approach is quite complex, but also the implementation of decompressed networks requires high-precision arithmetic units [13].

In this work, we develop a decoding-conscious weight representation method not only to highly compress the network but also to implement it very efficiently in real-time. The proposed algorithm trains the network so that the weights are represented using a structured sparse ternary format. This format allows +1 or -1 only at specified locations, while most of the values are pruned to zero. The network can achieve the compression ratio of almost up to x32, but the performance of the network is much better than the binary or XNOR networks. The hardware for inference contains a small look-up table for decompressing the code, but the procedure is very simple and deterministic. The data-path needs a reduced number of arithmetic units because most of the weights are pruned to zero. The indexing addresses can be easily interpreted to corresponding weights. Also, this method has a good scalability because the look-up table size is independent of the network complexity. However, training the structured sparsity network is more difficult than optimizing the conventional ternary valued networks. We use batch normalization and weight normalization techniques to mitigate the performance degradation. Also gradual pruning technique is applied for a large-sized network to improve the performance. The proposed scheme was evaluated on FCDNN, VGG-9, and AlexNet and obtained the compression rate between x23 and x32.

The rest of this paper is organized as follows. Section II

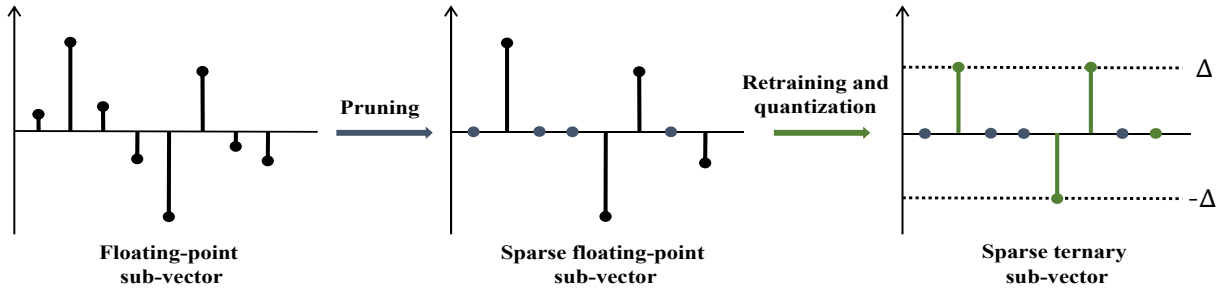


Fig. 1: The process of sub-vector structure sparse ternary quantization when  $(N, K)$  is  $(8, 4)$ .

TABLE I: The number of table entries ( $T$ ), the table size( $S_T$ ) and the length address ( $I$ ) for one sub-vector.

Codes	$T$	$S_T$ (KB)	$I$ (bits)
(16, 4)	34113	136.452	16
(16, 3)	4993	19.972	13
(16, 2)	513	2.052	10
(8, 2)	129	0.258	8
(8, 1)	17	0.034	5
(4, 1)	9	0.009	4

presents the proposed structured sparse ternary networks. In Section III, the network training method is presented. The effects of DNN normalizers and gradual pruning are also explained. Experimental results are shown in Section IV. Concluding remarks follow in Section V.

## II. STRUCTURED SPARSE TERNARY QUANTIZATION

### A. Review of ternary quantization

Ternary quantization represents a weight of a DNN using only +1, 0, and -1, and can achieve good performance by retraining [3], [6], [7]. In this case, one ternary weight is represented by 2 bits, thus a compression ratio of x16 can be obtained when compared with the 32-bit floating-point format. However, since only 3 levels (+1, 0, -1) are used, the information that can be represented with 2-bit per weight is not fully utilized. In addition, the ternary optimization results show that very high portion, about 85%, of the weights are zero, which implies the possibility of additional compression [3].

### B. Sub-vector structured sparsity

The proposed structured ternary quantization divides a weight into many one-dimensional sub-vectors with the size of  $N$ , and each sub-vector is represented by a ternary vector with a limited number,  $K$ , of +1 or -1. We first prune each sub-vector of the floating-point weight matrix to have only  $K$  non-zero values. Then, quantization and retraining are performed with pruned weights kept to zero. By the retraining and quantization, the number of non-zero values

can be decreased. The process is described in Fig. 1 when  $(N, K)$  is  $(8, 4)$ . In this figure, the final vector is determined as  $[0, +1, 0, 0, -1, +1, 0, 0]$ , and contains only 3 non-zero values

This structure can reduce the weight storage using a look-up table and indexing addresses. For example, the  $(4, 1)$  structured sparse coding denotes that the sub-vector length is 4 and only one position is allowed to be +1 or -1. Since the number of sub-vectors satisfying this condition is 9 including  $(0, 0, 0, 0)$ ,  $(0, 0, 0, +1)$ ,  $(0, 0, 0, -1)$ , ..., and  $(-1, 0, 0, 0)$ , the sub-vector index can be represented in 4 bits. As a result, the number of bits for  $(4, 1)$  structured ternary encoding is just 1-bit per weight excluding the memory for look-up table. In this structured sparse ternary encoding scheme, the sub-vector size,  $N$ , needs to be limited, such as 8 or 16, because the look-up table size increases as  $N$  grows. The proposed structured sparse ternary network needs a look-up table and indexing addresses. All possible sub-vectors are stored in the look-up table. Since the number of look-up table entries,  $T$ , is  $\sum_{i=0}^K \binom{N}{i} 2^i$ , the table occupies  $2NT$  bits and the indexing address of the sub-vector demands  $\lceil \log_2 T \rceil$  bits. For example, if  $(N, K)$  is  $(16, 4)$ , the total number of table entries is 34,113 and the address length is 16 bits. Also, if  $(N, K)$  is  $(8, 1)$ , the address length becomes 5 bits and the table size is just 34 Bytes. Since the table size is relatively small, the weight storage can be further reduced when compared to the conventional ternary coding. The proposed method only performs table indexing without any complicated decoding process, thus there is little decoding overhead. TABLE I shows the address length and the table size for each code employed for the experiments.

We only compress the weight matrix of fully-connected layers in FCDNNs and CNNs. Since fully-connected layers of large-sized CNNs usually consume over 90% of the weight storage, the weight matrix compression is important not only for FCDNNs but also for CNNs.

The propagation of a fully-connected layer can be represented with a matrix  $W$  as

$$\mathbf{y}_{k+1} = \phi_{k+1}(\mathbf{W}_{k+1}\mathbf{y}_k + \mathbf{b}_{k+1}), \quad (1)$$

where a row corresponds to connections for a single output neuron, and a column corresponds to connections from a single input neuron. We group the weights in the same column of  $W$  because the sub-vectors in this direction are less correlated.

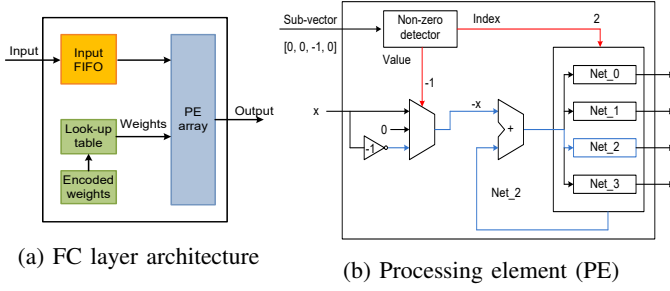


Fig. 2: Architecture of a structured sparse FC layer. (a) Overall architecture. (b) Structure of a processing element when  $(N, K)$  is  $(4, 1)$ .

The performance of row/column sub-vector structured sparse networks is compared in Section IV.

### C. Hardware design

The structured sparse ternary matrix can easily be decoded by employing table look-up operations. The overall architecture for a FC layer can be designed as shown in Fig. 2a. A sub-vector index is converted to an uncompressed weight sub-vector simply by indexing the look-up table. Since we drastically reduce the size of encoded weights and the look-up table, as shown in Section IV, the power consumption by external memory access can be eliminated or reduced with a small weight decoding overhead.

In addition, unnecessary computation can easily be removed with the column sub-vector structured sparsity. The column sub-vector approach also supports the outer-product based implementation, which is advantageous to parallel processing [4], [5]. In the outer product approach, all PEs receive the same input and they conduct outer-product with the consecutive weights. If we choose the column sub-vector based structure, zero weights can be ignored efficiently. Fig. 2b shows the structure of PE with an example flow when  $(N, K)$  is  $(4, 1)$ . One PE has only  $K$  arithmetic units and processes  $N$  outputs. For example, when the weight vector is  $[0, 0, -1, 0]^T$ , the non-zero detector block finds the non-zero value  $-1$  and the index of non-zero as 2. The non-zero value detection can easily be done because the length of a sub-vector is small, which is from 4 to 16 in our experiments. Since the weights are ternary quantized, the value just decides whether the input  $x$  is added, subtracted, or ignored. The index decides which register is activated. The selected register accumulates the input while others with 0 valued weights keep the current values.

## III. TRAINING OF STRUCTURED SPARSE TERNARY NETWORKS

Structured sparse ternary coding applies a constraint on the maximum number of non-zero weights in addition to ternary quantization. These constraints can incur performance loss. This section explains the structured sparse training method.

### - Masking weights:

$$\mathbf{W}_l = \mathbf{W}_{l, \text{trained}} \odot \mathbf{M}_l$$

### - Quantization step size determining:

$$\Delta_l = Qstep(\mathbf{W}_l) = \underset{\Delta}{\operatorname{argmin}} \sum_{w_{ij} \in \mathbf{W}_l} (Q(w_{ij}, \Delta_l) - w_{ij})^2$$

### - Quantized weights:

$$w_{ij}^{(q)} = Q(w_{ij}, \Delta) = \operatorname{sgn}(w_{ij}) \cdot \Delta \cdot \min\left(\left\lfloor \frac{|w_{ij}|}{\Delta} + 0.5 \right\rfloor, \frac{P-1}{2}\right)$$

### - Loss calculation:

$$\mathbf{net}_l = \mathbf{W}_l^{(q)} \mathbf{y}_l + \mathbf{b}_l$$

$$\mathbf{y}_{l+1} = \phi_l(\mathbf{net}_l)$$

$$E = - \sum_{(t, \mathbf{y}_L)} t \log y_L$$

### - Weights update:

$$w_{ij, \text{new}} = w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} \cdot m_{ij}$$

$$w_{ij, \text{new}}^{(q)} = Q(w_{ij, \text{new}}, \Delta_{\text{new}})$$

Fig. 3: The retraining algorithm of structured sparse fixed-point network is summarized.  $\Delta$  is the quantization step size,  $P$  is the number of quantization points,  $l$  denotes the layer, which is from 1 to  $L$ ,  $\mathbf{y}_l$  is the output vector of layer  $l$ ,  $\phi_l()$  is the activation function,  $E$  is the loss,  $t$  is one-hot encoded label vector, and  $\alpha$  is the learning rate. Floating-point weights  $\mathbf{W}$  and fixed-point weights  $\mathbf{W}^{(q)}$  are kept pruned by masking matrix  $\mathbf{M}$ .

### A. Structured pruning and quantization

The network is trained in floating-point first. Then, we simultaneously conduct the structured pruning and quantization, and then retrain the network. Each sub-vector of the floating-point weight matrix is pruned in order of magnitude so that every sub-vector only has  $K$  non-zero elements. The masking matrix  $\mathbf{M}$  is a Boolean matrix showing the pruned locations. In other words, if  $w_{ij}$  is pruned, then  $m_{ij}$  becomes 0, otherwise 1. The quantization step size  $\Delta$  is calculated using the pruned weight matrix instead of the original one. Forward propagation procedure is the same with the previous retraining based quantization algorithm [3].

We modify the backpropagation algorithm to maintain the structured sparsity during retraining. The gradients for pruned connections are removed by using the masking matrix. This algorithm keeps pruned weights unchanged, while updating the quantized weights. The overall algorithm is illustrated in Fig. 1 and also summarized in Fig. 3.

### B. Weight distribution normalization with DNN normalizers

In order to reduce the performance degradation of the structured sparse ternary coding, we apply the batch normalization

and weight normalization techniques.

1) *Batch normalization*: Batch normalization (BN) is widely used for training of FCDNNs and CNNs [14]. BN mitigates the gradient descent problem and acts as a regularizer when the training data is large enough. It normalizes the output neurons of DNNs in the same mini-batch during the forward and backward passes. In each training batch, BN renders the outputs of each neuron follow the Gaussian distribution.

2) *Weight normalization*: Weight normalization (WN) uses a simple reparameterization of the weights that accelerates the DNN training [15]. During training, each weight vector  $w$  is rescaled to  $v$ , which has unit  $L_2$ -norm. The loss is calculated with  $v$  and gradients are applied to  $w$ . For the inference,  $L_2$ -normed weights are used instead of the original ones.

### C. Gradual pruning and quantization

If many connections are pruned in a single step, it is difficult to compensate for the loss by retraining. Gradual pruning can alleviate this problem, which repeats pruning and retraining gradually from low sparsity to high sparsity. At each iteration, a small number of connections are pruned to preserve the performance. Then, after retraining, an increased number of weights are forced to zero and retraining is performed again. In our scheme, we combine the gradual pruning and quantization. At the first iteration, the network is retrained according to the proposed algorithm with a low sparse  $(N, K)$ , which means a large  $K$ . Floating-point weights  $W$  and fixed-point weights  $W^{(q)}$  are both retrained at this iteration. At the next iteration, instead of  $W^{(q)}$ ,  $W$  is pruned with higher sparsity by decrementing  $K$ . Further, we determine the step size  $\Delta$  using  $W$  because the weight values change much during retraining. The effect of adapting the step size to the changed weights is shown in [16]. After the network is pruned for the target sparsity, the final  $W^{(q)}$  is used for the inference.

## IV. EXPERIMENTAL RESULTS

### A. InfMNIST

The MNIST is a handwritten digit recognition dataset that consists of  $28 \times 28$  greyscale images. The InfMNIST dataset is derived from the MNIST using pseudo-random deformations and translations [17]. The training set is composed of 1M examples among the 8M sample data. The test set is the same with the original MNIST dataset, and 50K examples in the training set are used for validation. We train the networks using ADAM [18] optimizer. The learning rate decreases from  $1e-3$  to  $1.6e-5$  with a factor of 0.2 when the validation does not show improvements for 4 consecutive evaluations. The experimental results are the averages of 5 experiments with different random seeds. The network configuration is as shown below.

$$Input - Hidden1(1024) - Hidden2(1024) - 10Softmax, \quad (2)$$

The weight matrix of the output layer is quantized, but not pruned because the size is small. Also, biases and normalization parameters are kept in high precision. The performances

TABLE II: Miss classification rate(MCR(%)) on the test set with InfMNIST example. ‘Baseline’ means the networks are trained without any DNN normalizer.

	Baseline	BN	WN
Float network	1.03	0.72	0.77
Ternary network	1.30	0.86	1.05
Col sub-vector (16,3)	1.60	0.92	1.00
Row sub-vector (16,3)	2.62	0.93	1.03

according to the direction of the sub-vectors and normalizers are shown in TABLE II. In all experiments, the sub-vector length  $N$  is 16 and the maximum number of non-zero in a sub-vector  $K$  is 3. The ternary network is retrained using the algorithm shown in [3]. BN and WN show very high accuracy on the floating-point network because they act as regularizers. Also, BN and WN improve the performance of ternary and structured sparse networks. These results show that the normalizers are effective in alleviating the performance degradation due to the structured sparsity constraint. As we discussed in Section II, the row sub-vector structure results in high error rate. With BN and the column sub-vector structure, we obtain x1.7 times of weight storage compression with 0.06% miss classification rate (MCR) loss when compared to the unconstrained ternary network.

### B. CIFAR-10

The CIFAR-10 dataset includes examples from ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The training set consists of 50K  $32 \times 32 \times 3$  RGB samples and the test set contains 10K samples. We use 10% of the training set as the validation set. For data augmentation, training data are horizontally flipped with a probability of 50% at every epoch. Also, global contrast normalization(GCN) is applied to all images. The training procedure is the same with the InfMNIST task. We use VGG-9, which is modified to accommodate CIFAR-10 input data. The network configuration is as shown below.

$$Input - (2 \times 128C3) - MP2 - (2 \times 256C3) - MP2 \\ - (2 \times 512C3) - MP2 - (2 \times 1024FC) - 10Softmax, \quad (3)$$

This network demands 54.8MB for weight storage with the floating-point format and 3.45MB when the network is ternary quantized. For the experiments, BN is applied to the CONV layers and different normalizers are applied to the FC layers. The performances of the networks are shown in TABLE III.  $(N, K)$  is (16, 4) for the structured sparse networks. Floating-point networks show the lowest error rate when normalizers are not applied. Although ternary quantization decreases the performance a lot, the column sub-vector structured sparse

TABLE III: MCR(%) of CIFAR-10 classification task.

	Baseline	BN	WN
Float network	8.00	8.55	9.16
Ternary network	<b>9.36</b>	8.94	9.74
Col sub-vector (16,4)	9.81	<b>8.92</b>	<b>9.41</b>
Row sub-vector (16,4)	9.75	8.97	9.77

TABLE IV: MCR and the weight storage comparison with various  $(N, K)$  for CIFAR-10.  $N$  is sub-vector length, and  $K$  is the number of non-zeros in a sub-vector.

$(N, K)$	MCR(%)		Weight storage(MB)	Compression ratio
	BN	WN		
Float	8.55	9.16	54.804	X1
Ternary	8.94	<b>9.74</b>	3.453	X15.87
(16,4)	8.92	9.87	2.433	X22.52
(8,2)	<b>8.88</b>	9.78	2.301	X23.81
(4,1)	9.02	10.01	2.301	X23.81
(16,3)	9.35	10.11	2.097	X26.14
(16,2)	8.93	10.36	1.874	X29.25
(8,1)	8.91	10.32	<b>1.869</b>	<b>X29.32</b>

network trained with BN shows quite good performance. Further, our scheme shows better results than the ternary network when BN and WN are applied. This is because proper pruning can prevent over-fitting.

TABLE IV shows the performance and the weight storage compression ratio of networks with various choices of  $(N, K)$ . The column sub-vector structure is applied to all experiments. When the networks have the same sparsity, the constraint is stronger when  $N$  is small, which can decrease the performance. However, if  $N$  is large, the weight storage occupied by the table increases. By applying BN and the (8, 1) structured sparse network, we achieve x29.32 weight storage compression with MCR loss of 0.36% compared to the float-point network.

### C. ImageNet

The ImageNet is a 1000 objects classification problem. We train AlexNet with ImageNet ILSVRC-2012 dataset, which has 1.2M training data and 50K validation data. BN is applied to all CONV layers instead of skipping response normalization. Training is performed with the Matconvnet framework [19].

AlexNet has 61M parameters, of which 54.5M parameters are devoted to the FC layers. Therefore, it is important to compress FC layers to reduce the weight storage. The result of retraining to have the structured sparsity for various kinds of  $(N, K)$  is shown in TABLE V. Top-1 and top-5 MCR of the floating-point network are 41.73% and 18.94%,

TABLE V: Top-1 error rate, top-5 error rate(%) and the weight storage comparison with various  $(N, K)$  for ImageNet classification problem.

$(N, K)$	Top-1	Top-5	Weight storage(MB)	Compression ratio
Float	41.73	18.94	232.61	X1
Fixed	42.37	<b>19.50</b>	16.28	X14.29
(16,4)	<b>42.17</b>	19.89	9.92	X23.45
(8,2)	42.58	19.79	9.78	X23.78
(4,1)	43.04	20.00	9.78	X23.78
(16,3)	42.94	20.08	8.78	X26.49
(16,2)	43.51	20.52	7.35	X31.66
(8,1)	43.64	20.66	<b>7.34</b>	<b>X31.68</b>

TABLE VI: Comparison of the error rate(%) between the gradual scheme and direct pruning. Note that ‘Direct’ means that the network is pruned by target sparsity at once. All results are evaluated after the retraining is conducted.

$(N, K)$	Gradual		Direct	
	Top-1	Top-5	Top-1	Top-5
(8,4)	42.24	19.75	42.24	19.75
(8,3)	<b>42.05</b>	<b>19.68</b>	42.65	19.93
(8,2)	<b>42.48</b>	19.92	42.58	<b>19.79</b>
(8,1)	<b>43.07</b>	<b>20.26</b>	43.64	20.66

respectively. Fixed-point networks are quantized with 8-bit precision for CONV layers and 2-bit precision for FC layers. The unstructured fixed-point network shows 42.37% top-1 MCR and 19.50% top-5 MCR. For example, when  $(N, K)$  is (16, 4), the weight storage compression of x22.64 is obtained compared to the floating-point network. In this case, top-1 error increases 0.43%. Performance degradation increases as the sparsity grows. By allowing a top-1 error increase of up to 2% over the floating-point network, x31.68 compression ratio is achievable.

We further perform the gradual pruning to increase the performance of the structured sparse ternary network. The results are shown in TABLE VI. For every iteration,  $K$  is decremented by 1. Gradually pruned network shows 43.07% top-1 MCR, which is lower than that of the directly pruned network by 0.57%. Finally, we obtain x31.68 weight storage compression ratio with a top-1 MCR increase of 1.34% over the floating-point network.

We compare our model with other fixed-point networks on TABLE VII. The baseline is the floating-point network trained using Matconvnet framework. XNOR-Net achieves a compression ratio of x32 through binary quantization, but the error increases sharply. Ternary weight network (TWN) [6]

TABLE VII: Comparison of networks in the error rate(%) and the weight storage compression ratio. Results are on ImageNet data with AlexNet.

	Top-1	Top-5	Weight storage(MB)	Ratio
Float(baseline)	41.7	18.9	232.6	X1
XNOR-Net	55.8	30.8	7.3	X32
TWN	45.5	23.2	17.0	X14
TWN_V2	<b>42.5</b>	20.3	17.0	X14
Deep Compression	42.8	<b>19.7</b>	<b>6.9</b>	<b>X35</b>
<b>Ours</b>	43.1	20.3	7.3	X32

and TWN V2 [7] quantize weights to ternary values. TWN V2 shows 42.5% Top-1 error rate, which is the lowest among the fixed-point networks. Deep Compression uses the relative indexed CSC/CSR format and Huffman coding to compress the sparse fixed-point network and reduces the weight storage by x35 while maintaining the performance. However, TWN V2 and Deep Compression have disadvantages of hardware implementation. Since each quantization point has a different step size, high-precision arithmetic operations are needed. On the other hand, XNOR-Net, TWN, and ours can substitute the multiply-accumulate operations to simple logical or accumulate operations [4], [5], [8]. Further, Deep Compression needs additional decoding units [13]. Considering the complexity and overhead, our coding scheme is very advantageous to hardware implementations.

## V. CONCLUDING REMARKS

We presented a structured sparse ternary coding scheme for low-energy hardware implementation of deep neural networks. The proposed method compresses a matrix for a fully-connected layer by decomposing it into sub-vectors and allowing only a limited number of non-zero values in each sub-vector. The decoding is conducted simply by consulting a look-up table. Batch normalization and gradual pruning techniques are employed to mitigate the performance degradation due to structured sparsity. We can reduce the weight storage for VGG-9 and AlexNet to 1.87MB (x29 compression) and 7.34MB (x32 compression) with only small accuracy loss. This research is useful for the implementation of large-sized DNNs on resource-limited hardware.

## ACKNOWLEDGMENT

This work was supported in part by the Brain Korea 21 Plus Project and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2015R1A2A1A10056051).

## REFERENCES

[1] Christopher M Bishop, *Neural networks for pattern recognition*, Oxford university press, 1995.

[2] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Nguyen Patrick, Tara N Sainath, et al., “Deep neural networks for acoustic modeling in speech recognition,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82, 2012.

[3] Kyuhyeon Hwang and Wonyong Sung, “Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1,” in *Signal Processing Systems (SIPS), 2014 IEEE Workshop on*. IEEE, 2014, pp. 1–6.

[4] Jonghong Kim, Kyuhyeon Hwang, and Wonyong Sung, “X1000 real-time phoneme recognition vlsi using feed-forward deep neural networks,” in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 7510–7514.

[5] Jinhwan Park and Wonyong Sung, “Fpga based implementation of deep neural networks using on-chip memory only,” in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1011–1015.

[6] Fengfu Li, Bo Zhang, and Bin Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711*, 2016.

[7] Chenzhao Zhu, Song Han, Huizi Mao, and William J Dally, “Trained ternary quantization,” in *ICLR*, 2017.

[8] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[9] Song Han, Huizi Mao, and William J Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in *ICLR*, 2016.

[10] Wonyong Sung, Sungho Shin, and Kyuhyeon Hwang, “Resiliency of deep neural networks under quantization,” *arXiv preprint arXiv:1511.06488*, 2015.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[12] Karen Simonyan and Andrew Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *ICLR*, 2015.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally, “Eie: efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[14] Sergey Ioffe and Christian Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015, pp. 448–456.

[15] Tim Salimans and Diederik P Kingma, “Weight normalization: A simple reparameterization to accelerate training of deep neural networks,” in *Advances in Neural Information Processing Systems*, 2016, pp. 901–901.

[16] Sungho Shin, Yoonho Boo, and Wonyong Sung, “Fixed-point optimization of deep neural networks with adaptive step size retraining,” in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, 2017.

[17] “Projects:infinnism,” <http://leon.bottou.org/projects/infinnism>, Accessed: 2017-05-01.

[18] Diederik P. Kingma and Jimmy Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[19] Andrea Vedaldi and Karel Lenc, “Matconvnet: Convolutional neural networks for matlab,” in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 689–692.