Compact and Computationally Efficient Representation of Deep Neural Networks

Simon Wiedemann
Fraunhofer HHI
10587 Berlin, Germany
simon.wiedemann@hhi.fraunhofer.de

Klaus-Robert Müller
TU Berlin, Korea University & MPII
10587 Berlin, Germany
klaus-robert.mueller@tu-berlin.de

Wojciech Samek
Fraunhofer HHI
10587 Berlin, Germany
wojciech.samek@hhi.fraunhofer.de

Abstract

At the core of any inference procedure in deep neural networks are dot product operations, which are the component that require the highest computational resources. One common approach to reduce the complexity of these operations is to prune and/or quantize the weight matrices of the neural network. Usually, this results in matrices whose entropy value is low, as measured relative to the maximum likelihood estimate of the probability mass distribution of it's elements. In order to efficiently exploit such matrices one usually relies on, inter alia, sparse matrix representations. However, most of these common matrix storage formats make strong statistical assumptions about the distribution of the elements in the matrix, and can therefore not efficiently represent the entire set of matrices that exhibit low entropy statistics (thus, the entire set of compressed neural network weight matrices). In this work we address this issue and present new efficient representations for matrices with low entropy statistics. We show that the proposed formats can not only be regarded as a generalization of sparse formats, but are also more energy and time efficient under practically relevant assumptions. For instance, we experimentally show that we are able to attain up to x16 compression ratios, x1.7 speed ups and x20 energy savings when we convert the weight matrices of state-of-the-art networks such as AlexNet, VGG-16, ResNet152 and DenseNet into the new representations.

1 Introduction

Deep neural networks [10, 16] became the state-of-the-art in many fields of machine learning. However, most deep neural network models require the computation of many dot product operations between large matrices when performing inference. This requires a great amount of computational resources which difficult their deployment into resource constrained devices.

This fact motivated an entire research field of model compression [1] that aimed to reduce the complexity of inference of deep neural networks. For instance, some of the most popular techniques include sparsification of the network [3, 9, 8, 14] or quantization of the weight elements [19, 12]. These approaches aim to reduce both storage and execution complexity of the dot product oprations involved in the inference procedure, by either leveraging on the sparsity of the weight matrices, e.g., by representing them in one of the sparse storage formats [15, 4] and performing the dot products accordingly, or by converting the element values into low bit-length numerical representations.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

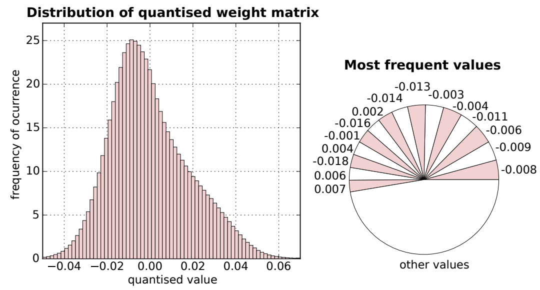

Figure 1: Distribution of the weight matrix of the last layer of the VGG-16 neural network [17] after quantization. The respective matrix is 1000×4096 dimensional, transforming the 4096 last-layer features onto 1000 output classes. We applied an uniform quantizer over the range of values, with 2^7 quantization points, which resulted in no loss of accuracy on the classification task. Left: Probability mass distribution. Right: Frequency of appearance of the 15 most frequent values.

However, highest compression ratios and execution efficiencies can be achieved when one employs compression schemes that either implicitly [7, 11, 13] or explicitly [2, 18, 5] aim to minimize the entropy of the weight elements relative to the maximum likelihood estimate of their probability mass distribution.

However, weight matrices that exhibit low entropy statistics are not necessarily sparse, neither the cardinality of the elements has to be low. For instance, figure 1 plots the distribution of the weight elements of the last classification layer of VGG-16 [17] (1000×4096 dimensional matrix), after having applied uniform quantization on the weight elements. We stress that the prediction accuracy and generalization of the network was not affected by this operation. As we can see, on the one hand, the distribution of the compressed layer does not satisfy the sparsity assumption, i.e., there is not one particular element (such as 0) that appears specially frequent in the matrix. On the other hand, we can also not take advantage of the fact that we could trivially compress each element of the original dense representation down to 7 bits (because we only have at most 2^7 unique elements present in the matrix), since most conventional hardware do not support multiplications on mixed-precission numerical representations (recall, that in this case the activation values would be still represented by single precission floating point values). Moreover, the entropy of the matrix is about $H = 5.5$, which is smaller than 7 bits, thus indicating that even higher efficiency gains may be obtained. Although [7] proposed an optimal entropy coder for representing these type of matrices, specialized hardware is needed in order to efficiently run inference under the resulting representation [6]. Hence, under our knowledge, there is currently no simple representation that can be applied on this type of matrices, that is able to simultaneously achieve high compression ratios and low computational complexity relative to the dot product operation.

2 The compressed entropy row (CER) and compressed shared weight row (CSER) representations

Notice, that we expect the ratio $\bar{k}/n$ between the average number of unique elements $\bar{k}$ that appear in a particular row (or column) and the respective number of columns (or rows) n to be *low*, for matrices that exhibit low entropy statistics. The intuition behind this is that we can interpret the number 2^H as the effective number of unique elements that a random variable with entropy value H outputs. Indeed, only 15 distinct values dominate the entries of the weight matrix displayed in figure 1, which is only 1.5% of the number of columns of the matrix. Hence, in this work, we propose two new matrix representations that exploit this property, which we named *compressed shared weight row* (CSER) and *compressed entropy row* (CER). Both apply the same principles, but differ only slightly on their statistical assumptions.

The CER and CSER representations are able to save significant amounts of storage requirements by leveraging on the fact that we only need to store the unique elements that appear on the matrix rows *once*. In contrast, sparse storage formats store the same non zero values multiple times, inducing high redundancies in their final representations. Moreover, we can design efficient dot product algorithm by implicitly encoding the distributive law of multiplications in the CER and CSER representations. By

Table 1: Storage, number of operations, time and energy gains (reported in the same order in the table cells) of performing a matrix-vector multiplication on the weight matrices of different state-of-the-art neural networks. Each weight matrix has been compressed down to 7 bits and, subsequently, converted into the different matrix storage formats. The performance gains are relative to the original, uncompressed dense representation. The accuracy is measured with regard to the validation set (in parenthesis we show the accuracy of the original model).

Size [MB], #ops [G], time [s], energy [J]	Accuracy [%]	dense	CSR	CER	CSER
VGG16	68.51 (68.71)	553.43, 15.08 3.37, 2.70	x0.71, x0.88 x0.85, x0.76	x2.11, x1.40 x1.27, x2.37	x2.11, x1.39 x1.29, x2.38
ResNet152	78.17 (78.25)	240.77, 10.08 2.00, 1.92	x0.76, x0.93 x0.93, x1.25	x2.08, x1.42 x1.30, x3.73	x2.10, x1.41 x1.31, x3.74
DenseNet	77.09 (77.12)	114.72, 7.14 1.53, 0.51	x1.04, x1.11 x1.10, x1.95	x2.74, x1.66 x1.43, x6.40	x2.79, x1.65 x1.45, x6.57

incorporating the distributive law we can not only save a great number of multiplications, but also a great number of expensive read/write operations associated to them. The particular encoding/decoding procedure as well as the associated dot product algorithms are described more extensively in our main manuscript [20]. In the appendix we show an example and also present the respective pseudocodes.

We can show that the storage requirement as well as the dot product complexity (as measured relative to each element in the matrix) of the CSER and CER representations asymptotically fulfill

$$\text{complexity} = \mathcal{O}((1-p_0)\log_2 n) + \mathcal{O}\left(\frac{\bar{k}}{n}\right) + \mathcal{O}\left(\frac{1}{n}\right) \tag{1}$$

where p_0 is the maximum likelihood estimate of the probability mass distribution of the most frequent value, n the row size and $\bar{k}$ the average number of unique elements that appear in a row. Hence, equation (1) states that the efficiency of the CER and CSER representations depend partially on the sparsity level of the matrix (the term $(1-p_0)$) and the per row cardinalty of it's elements (the term $\bar{k}$). Consequently, the CER and CSER representations will result in being particularly efficient when both terms are simultaneously minimized. Notice, that both terms decrease as the entropy of the matrix statistics decreases. Also, as we mentioned in the introduction, most frameworks that achieve state-of-the-art compression ratios aim to directly minimize these two components simultaneously [7, 11, 13, 2, 18, 5]. In addition, (1) also states that the CER and CSER representations can efficiently exploit a wider set of matrices, in particular those matrices whose element statistics fail to be efficiently executed by the dense and sparse representations.

3 Experiments

We applied our representations on compressed weight matrices of deep neural networks and benchmarked their efficiency with regards to 4 metrics: 1) the storage requirement, 2) the total number of operations needed to perform a matrix-vector multiplication, 3) the respective time complexity and 4) the respective energy consumption. We run all our experiments on a conventional CPU hardware.

We were able to attain up to x16 compression ratios, x1.7 speed ups and x20 energy savings (compared to the original dense representation) after converting the weight matrices of the by [7] compressed AlexNet model[1] into the CER and CSER representations. These are significant gains compared to, e.g., the common CSR sparse format [15, 4] which attained x6, x1.7 and x5 gains respectively. We also benchmarked our matrix representations on networks that were compressed using lossy compression techniques that do not require to retrain the network in the process. This case is of particular interest, since in many real world scenarios one may have a full-sized model but no access to the training data. Also, in most cases, the statistics of the compressed weight matrices do not fulfill the common prior assumptions of most conventional matrix representations in such scenario (e.g., figure 1). In contrast, the CER and CSER are able to exploit the statistical properties present, and attain higher gains in efficiency throughout all four above mentioned benchmarks (see table 1).

[1] `https://github.com/songhan/Deep-Compression-AlexNet`

Acknowledgment

This work was supported by the Fraunhofer Society through the MPI-FhG collaboration project "Theory & Practice for Reduced Learning Machines". This research was also supported by the German Ministry for Education and Research as Berlin Big Data Center under Grant 01IS14013A and by the Institute for Information & Communications Technology Promotion and funded by the Korea government (MSIT) (No. 2017-0-01779 and No. 2017-0-00451).

References

[1] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv:1710.09282*, 2017.

[2] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Universal deep neural network compression. *CoRR*, abs/1802.02271, 2018.

[3] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*, pages 598–605, 1990.

[4] I. S. Duff. A survey of sparse matrix research. *Proceedings of the IEEE*, 65(4):500–535, 1977.

[5] M. Federici, K. Ullrich, and M. Welling. Improved Bayesian Compression. *ArXiv e-prints*, November 2017.

[6] 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 *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[7] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.

[8] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[9] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1, 1993.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[11] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv:1605.04711*, 2016.

[12] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[13] C. Louizos, K. Ullrich, and M. Welling. Bayesian Compression for Deep Learning. In *Advances in Neural Information Processing Systems*, pages 3290–3300, 2017.

[14] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *34th International Conference on Machine Learning*, pages 2498–2507, 2017.

[15] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[16] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[18] K. Ullrich, E. Meeds, and M. Welling. Soft Weight-Sharing for Neural Network Compression. *ArXiv e-prints*, February 2017.

[19] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *NIPS'11 Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

[20] Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Compact and computationally efficient representation of deep neural networks. *CoRR*, abs/1805.10692, 2018.

A Example of how the CER and CSER formats compare to the dense and CSR matrix representations

Consider the following matrix

$$M = \begin{pmatrix} 0 & 3 & 0 & 2 & 4 & 0 & 0 & 2 & 3 & 4 & 0 & 4 \\ 4 & 4 & 0 & 0 & 0 & 4 & 0 & 0 & 4 & 4 & 0 & 4 \\ 4 & 0 & 3 & 4 & 0 & 0 & 0 & 4 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 4 & 4 & 4 & 0 & 3 & 4 & 4 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 & 4 & 0 & 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Now assume that we want to: 1) store this matrix with the minimum amount of bits and 2) perform the dot product with a vector $a \in \mathbb{R}^{12}$ with the minimum complexity.

A.1 Storage requirements

Dense format: The dense representation would trivially store the elements of M in a 5×12 long array (in addition to it's dimensions $m = 5$ and $n = 12$).

Sparse format: Sparse formats attempt to attain higher compression ratios by not storing the 0 values in their matrix representation. In particular, the *Compressed Sparse Row* (or CSR in short) format stores the values of the matrix in the following way:

- Scans the non-zero elements in row-major order (that is, from left to right, up to down) and stores them in an array (which we denote as W).
- Simultaneously, it stores the respective column indices in another array (which we call $colI$).
- Finally, it stores pointers that signalize when a new row starts (we denote this array as $rowPtr$).

Hence, the matrix M would take the form

$$\begin{aligned} W :&[3,2,4,2,3,4,4,4,4,4,4,4,4,3, \\ &4,4,2,4,4,4,3,4,4,4,4,4,4] \\ colI :&[1,3,4,7,8,9,11,0,1,5,8,9,11,0, \\ &2,3,7,9,3,4,5,7,8,9,1,2,5,7] \\ rowPtr :&[0,7,13,18,24,28] \end{aligned}$$

If we assume the same bit-size per element for all arrays, then the CSR data structure does not attain higher compression gains in spite of not saving the zero valued elements (62 entries vs. 60 that are being required by the dense data structure).

CER format: Firstly, notice that many elements in M share the same value. In fact, only the four values $\Omega = \{0,4,3,2\}$ appear in the entire matrix. Hence, it appears reasonable to assume that data structures that repeatedly store these values (such as the dense or CSR structures) induce high redundancies in their representation. Secondly, notice that different elements appear more frequent than others, and their relative relation does not change throughout the rows of the matrix, and the ordering. Concretely, we have a set of unique elements $\Omega = \{0,4,3,2\}$ which appear $P_{\#} = \{32,21,4,3\}$ times respectively in the matrix, and we obtain the same relative order of highest to lowest frequent value throughout the rows of the matrix. Hence, we can design an efficient data structure which leverages on both properties in the following way:

1. Store unique elements present in the matrix in an array in frequency-major order (that is, from most to least frequent). We name this array Ω.

2. Store respectively the column indices in row-major order, excluding the first element (thus excluding the most frequent element). We denote it as $colI$.
3. Store pointers that signalize when the positions of the next new element in Ω starts. We name it ΩPtr. If a particular pointer in ΩPtr is the same as the previous one, this means that the current element is not present in the matrix and we jump to the next element.
4. Store pointers that signalize when a new row starts. We name it $rowPtr$.

Hence, this new data structure represents matrix M as

$$\begin{aligned}
\Omega :&[0, 4, 3, 2]\\
colI :&[4, 9, 11, 1, 8, 3, 7, 0, 1, 5, 8, 9, 11, 0,\\
&3, 7, 2, 9, 3, 4, 5, 8, 9, 7, 1, 2, 5, 7]\\
\Omega Ptr :&[0, 3, 5, 7, 13, 16, 17, 18, 23, 24, 28]\\
rowPtr :&[0, 3, 4, 7, 9, 10]
\end{aligned}$$

Notice that we can uniquely reconstruct M from this representation. One can verify that indeed, the CER data structure only requires 49 entries (instead of 60 or 62) attaining as such a compressed representation of the matrix M.

CSER format: In some cases, it may well be that the probability distribution across rows in a matrix are not similar to each other. Hence, the second assumption in the CER data structure would not apply and we would only be left with the first one. That is, we only know that not many distinct elements appear per row in the matrix or, in other words, that many elements share the same value. The *compressed shared elements row* (or CSER in short) data structure is a slight extension to the previous CER representation. Here, we add an element pointer array, which signalizes which element in Ω the positions are referred to (we called it ΩI). Hence, the above matrix would then be represented as follows

$$\begin{aligned}
\Omega :&[0, 2, 3, 4]\\
colI :&[4, 9, 11, 1, 8, 3, 7, 0, 1, 5, 8, 9, 11, 0,\\
&3, 7, 2, 9, 3, 4, 5, 8, 9, 7, 1, 2, 5, 7]\\
\Omega I :&[3, 2, 1, 3, 3, 2, 1, 3, 2, 3]\\
\Omega Ptr :&[0, 3, 5, 7, 13, 16, 17, 18, 23, 24, 28]\\
rowPtr :&[0, 3, 4, 7, 9, 10]
\end{aligned}$$

Thus, for storing matrix M we require 59 entries, which is still a gain but not a significant one. Notice, that now the ordering of the elements in Ω is not important anymore.

A.2 Dot product complexity

We just saw that we can attain gains with regard to compression if we represent the matrix in the CER and CSER data structures. However, we can also devise corresponding dot product algorithms that are more efficient than their dense and sparse counterparts. As an example, consider only the scalar product between the second row of matrix M with an arbitrary input vector $a = [a_1\ a_2\ \dots\ a_{12}]^\top$. In principle, the difference in the algorithmic complexity arises because each data structure implicitly encodes a different expression of the scalar product, namely

$$\begin{aligned}
dense :&\ 4a_1 + 4a_2 + 0a_3 + 0a_4 + 0a_5 + 4a_6\\
&+ 0a_7 + 0a_8 + 4a_9 + 4a_{10} + 0a_{11} + 4a_{12}\\
CSR :&\ 4a_1 + 4a_2 + 4a_6 + 4a_9 + 4a_{10} + 4a_{12}\\
CER/CSER :&\ 4(a_1 + a_2 + a_6 + a_9 + a_{10} + a_{12})
\end{aligned}$$

For instance, the dot product algorithm associated to the dense format would calculate the above scalar product by

1. loading M and a.
2. calculating $4a_0 + 4a_1 + 0a_2 + 0a_3 + 0a_4 + 4a_5 + 0a_6 + 0a_7 + 4a_8 + 4a_9 + 0a_{10} + 4a_{11}$.

This requires 24 load (12 for the matrix elements and 12 for the input vector elements), 12 multiply, 11 add and 1 write operations (for writing the result into memory). We purposely omitted the accumulate operation which stores the intermediate values of the multiply-sum operations, since their cost can effectively be associated to the sum operation. Moreover, we only considered read/write operations from and into memory. Hence, this makes 48 operations in total.

In contrast, the dot product algorithm associated with the CSR representation would only multiply-add the non-zero entries. It does so by performing the following steps

1. Load the subset of $rowPtr$ respective to row 2. Thus, $rowPtr \rightarrow [7, 13]$.
2. Then, load the respective subset of non-zero elements and column indices. Thus, $W \rightarrow [4, 4, 4, 4, 4, 4]$ and $colI \rightarrow [0, 1, 5, 8, 9, 11]$.
3. Finally, load the subset of elements of a respective to the loaded subset of column indices and subsequently multiply-add them to the loaded subset of W. Thus, $a \rightarrow [a_0, a_1, a_5, a_8, a_{10}, a_{11}]$ and calculate $4a_0 + 4a_1 + 4a_5 + 4a_8 + 4a_9 + 4a_{11}$.

By executing this algorithm we would require 20 load operations (2 from the $rowPtr$ and 6 for the W, the $colI$ and the input vector respectively), 6 multiplications, 5 additions and 1 write. In total this dot product algorithm requires 32 operations.

However, we can still see that the above dot product algorithm is inefficient in this case since we constantly multiply by the same element 4. Instead, the dot product algorithm associated to, e.g., the CER data structure, would perform the following steps

1. Load the subset of $rowPtr$ respective to row 2. Thus, $rowPtr \rightarrow [3, 4]$.
2. Load the corresponding subset in ΩPtr. Thus, $\Omega Ptr \rightarrow [7, 13]$.
3. For each pair of elements in ΩPtr, load the respective subset in $colI$ and the element in Ω. Thus, $\Omega \rightarrow [4]$ and $colI \rightarrow [0, 1, 5, 8, 9, 11]$.
4. For each loaded subset of $colI$, perform the sum of the elements of a respective to the loaded $colI$. Thus, $a \rightarrow [a_0, a_1, a_5, a_8, a_{10}, a_{11}]$ and do $a_0 + a_1 + a_5 + a_8 + a_9 + a_{11} = z$.
5. Subsequently, multiply the sum with the respective element in Ω. Thus, compute $4z$.

A similar algorithm can be devised for the CSER data structure. One can find both pseudocodes in the appendix. The operations required by this algorithm are 17 load operations (2 from $rowPtr$, 2 from ΩPtr, 1 from Ω, 6 from $colI$ and 6 from a), 1 multiplication, 5 additions and 1 write. In total this are 24 operations.

Hence, we have observed that for the matrix M, the CER (and CSER) data structure does not only achieve **higher compression rates**, but it also attains **gains in efficiency** with respect to the dot product operation.

B CER and CSER dot product pseudocodes

Algorithms 1 and 2 show the pseudocodes of the dot product algorithm of the CER and CSER data structures.

Algorithm 1 CER dot product

procedure DOT_{cer}(M,X)
$\Omega, colI, wPtr, rowPtr \leftarrow M$
$N, L \leftarrow dim(X)$
$Y = 0 \in \mathbb{R}^{M \times L}$
for $l \leq L$ **do**
$r_{start} = 0$
$w_{start} = 0$
for $r_{idx} = 1 < len(rowPtr)$ **do**
$r_{end} \leftarrow rowPtr[r_{idx}]$
$y = 0$
$w_{count} = 1$
for $w_{idx} = r_{start} + 1 < r_{end} + 1$ **do**
$w_{end} \leftarrow wPtr[w_{idx}]$
$y' = 0$
for $i = w_{start} < w_{end}$ **do**
$I \leftarrow colI[i]$
$y' \leftarrow y' + X[I, l]$
if $w_{start} + 1 = w_{end}$ **then**
$y \leftarrow y + y' * \Omega[w_{count}]$
$w_{count} \leftarrow w_{count} + 1$
$w_{start} \leftarrow w_{end}$
$r_{start} \leftarrow r_{end}$
$Y[m, k] \leftarrow y$
return Y

Algorithm 2 CSER dot product

procedure DOT_{cser}(M,X)
$\Omega, colI, wI, wPtr, rowPtr \leftarrow M$
$N, L \leftarrow dim(X)$
$Y = 0 \in \mathbb{R}^{M \times L}$
for $l \leq L$ **do**
$r_{start} = 0$
$w_{start} = 0$
$w_{count} = 0$
for $r_{idx} = 1 < len(rowPtr)$ **do**
$r_{end} \leftarrow rowPtr[r_{idx}]$
$y = 0$
for $w_{idx} = r_{start} + 1 < r_{end} + 1$ **do**
$w_{end} \leftarrow wPtr[w_{idx}]$
$y' = 0$
for $i = w_{start} < w_{end}$ **do**
$I \leftarrow colI[i]$
$y' \leftarrow y' + X[I, l]$
$y \leftarrow y + y' * \Omega[wI[w_{count}]]$
$w_{count} \leftarrow w_{count} + 1$
$w_{start} \leftarrow w_{end}$
$r_{start} \leftarrow r_{end}$
$Y[m, k] \leftarrow y$
return Y