Huffman coding in neural network compression

Flavio Furia

AnacletoLab - Dipartimento di Informatica
Università degli Studi di Milano

5th PRIN Meeting, 22 September 2022

Multicriteria Data Structures and Algorithms:
from compressed to learned indexes, and beyond

Problem

AIM: Reduce the resources demanded by existing deep and convolutional Neural Networks, which are often large and overparameterized.

PROBLEM: To do this, we need to efficiently execute the vector-matrix/tensor product at each layer, in order to obtain an output.

Main state-of-the-art DNN compression Strategies

DNNs compression is facilitated by adding regularities in the layer weights:

- network pruning (neurons, layers, channels, filters, etc.)
- knowledge distillation
- sparse low-rank factorization [1]
- connection pruning
- quantization via weight sharing

We focus on the last two, to leave the NNs topology unaltered and make our approach more general.

Matrix-vector multiplication

Sparse matrix-vector multiplication (SpMV) can be done efficiently by exploiting sparsity.

Many techniques to perform SpMV, e.g.:

- Compressed Sparse Column/Row (CSC/CSR) [2, 3]
- Compressed Linear Algebra (CLA) [4]
- Compressed Shared Elements Row (CSER) [5]

Proposal: Realizing vector-matrix product via Huffman coding.

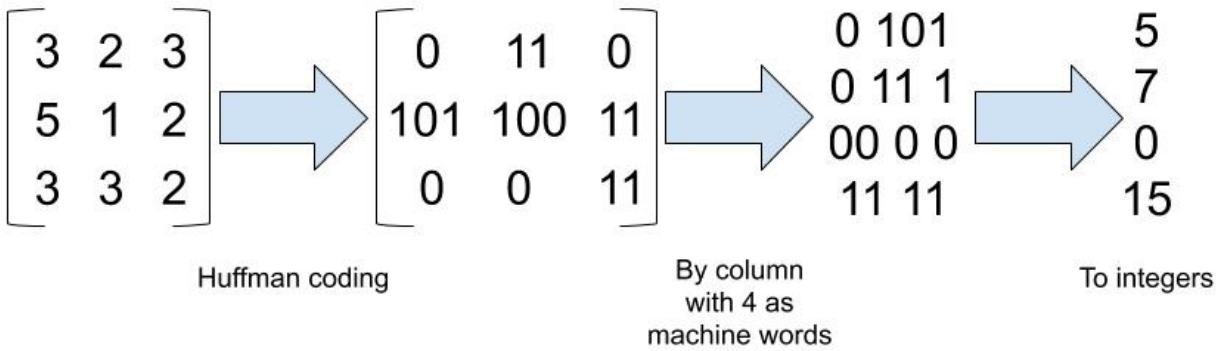
Strategy

- Consider a weight matrix $\boldsymbol{W} \in \mathbf{R}^{n \times m}$
- we apply sparsification and quantization to $\boldsymbol{W}$, obtaining a compressed version of the matrix, referred to as $\overline{\boldsymbol{W}}$
  - $\overline{\boldsymbol{W}}$ has k non-zero possible distinct values
  - we assume b bits are required to store one element of $\boldsymbol{W}$
  - b is also the memory word size
  - $s \in [0, 1]$ is the ratio of non-zero elements in $\overline{\boldsymbol{W}}$
- combination of
  - Lossy compression
  - Lossless representation

HAM - Huffman Address Map

AIM: exploit *quantization* in $\overline{\boldsymbol{W}}$

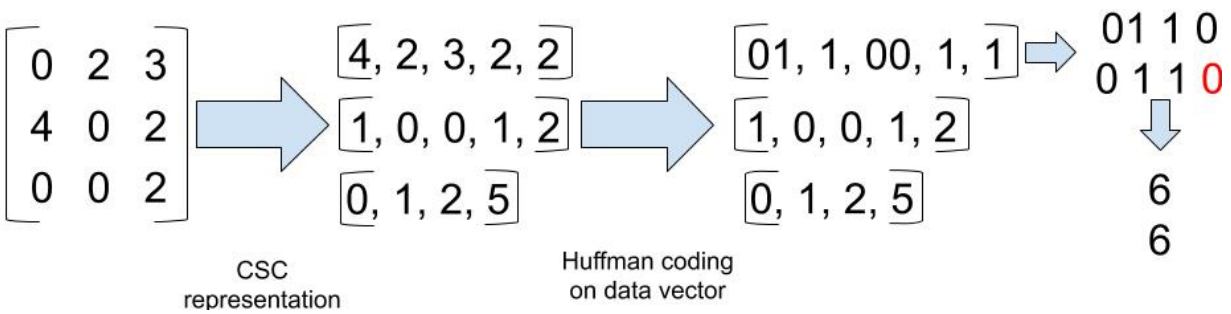
- Depending on k, the weights have a high frequency
- Following the suggestion in [6], we apply Huffman coding to quantized weights
- $\texttt{HAM}(\overline{\boldsymbol{W}})$ is obtained by coding weights in $\overline{\boldsymbol{W}}$ by column
- $\texttt{HAM}(\overline{\boldsymbol{W}})$ is split into $\left\lceil \frac{|\texttt{HAM}(\overline{\boldsymbol{W}})|}{b} \right\rceil$ integers
- To decompress we also need to store the inverse Huffman code



sHAM - Sparse Huffman Address Map

AIM: exploit *quantization* and *sparsity* in $\overline{\boldsymbol{W}}$

- Instead of coding the null weights, we exclude them by first representing $\overline{\boldsymbol{W}}$ in Compressed Sparse Column (CSC) format:
  - ***nz***, vector containing the nonzero values, listed by columns;
  - ***ri***, vector containing the row indices of elements in ***nz***;
  - ***cb***, where cb_i provides the number of nonzero elements in column i
- Compress ***nz*** with `HAM` format



Efficient Huffman Codes implementations

PROBLEM: Storing the Huffman tree and pointers is too demanding, we need a more efficient representation.

SOLUTION: Rely on Huffman codes implementations for which decoding does not require to access the tree, e.g. Canonical Huffman coding.

Several versions of Huffman codes have been implemented and tested on our `HAM` and `sHAM` formats.

We compared them with other common compact representations of sparse and quantized matrices (e.g. CSC), in terms of how many bits we need to store one element of $\overline{\boldsymbol{W}}$.

Huffman code with a lookup table

1st approach: Classical Huffman code with a lookup table

- read the bit stream by chunks of size t
- build a lookup table of size 2^t
- table is indexed by integer value of chunks
- one entry contains the symbols encoded in that chunk

Canonical Huffman Codes

2nd approach: Canonical Huffman coding

- One of the most efficient ways to avoid storing the Huffman tree relies on the employment of **Canonical Huffman Codes**.
- The **canonical property** tells us that code words with the same length are assigned values sequentially.
- this provides fast decoding, since we don't have to access the tree.

Several canonical variants of Huffman codes have been tested, in the end choosing the best one in terms of compactness and decoding time.

Example

symbol	*base*	*canonical*
0	0	1
8	100	011
4	1100	0001
5	1111	0010
6	1101	0011
9	1010	0100
10	1110	0101
3	10110	00000
7	10111	00001

Canonical property

Code words of the same length are assigned values sequentially

Some info about the first code word of each length and its associated symbol is enough to decode.

First canonical variant

2nd approach, a): read the bit stream a bit at a time

- start with an existing Huffman code
- re-arrange code words so that the canonical property holds
- build two arrays of length l_{max}, the max code word length, storing
  - the first code word of each length
  - the list of symbols with code words of each length

Second canonical variant

2nd approach, b): read the bit stream by chunks of l_{max} bits

- sort symbols by non-increasing order of their probability
- assign code words with increasing values with respect to their lengths
- build three arrays, respectively storing
  - the list of symbols we are encoding (total size k)
  - the index of the first symbol assigned to a code word of each length (total size l_{max})
  - the "left-justified" value, right-padded to l_{max}, of the first code word of each length (total size l_{max} plus a sentinel value)

Speed-up table

2nd approach, b): speedup the search of the first code word length

- choose a value $t \leq l_{max}$
- build a 2^t-sized table indexed by the integer value of chunks
- one entry contains the length of the first code word for a chunk, or where we should start the search
- l_{max} is upper bounded by $k - 1$, thus t should be chosen carefully
- a good empirical value is $t = \log(k - 1)$

HAM/sHAM – Comparison

- Canonical Huffman code space overhead (b word size):
  $B_k = k(k + 2\log k + b - 1) - 2\log k$ bits
  - $l_{max} = k - 1$ (worst case)
  - $t = \log l_{max}$
- Given an $n \times m$ with k distinct weights (ψ: # bits per element):

  Fact 1 (worst case): $\psi_{\texttt{HAM}} \leq 1 + \log k + \frac{B_k}{nm}$
- Given an $n \times m$ with k distinct weights and a non-zero ratio of s:

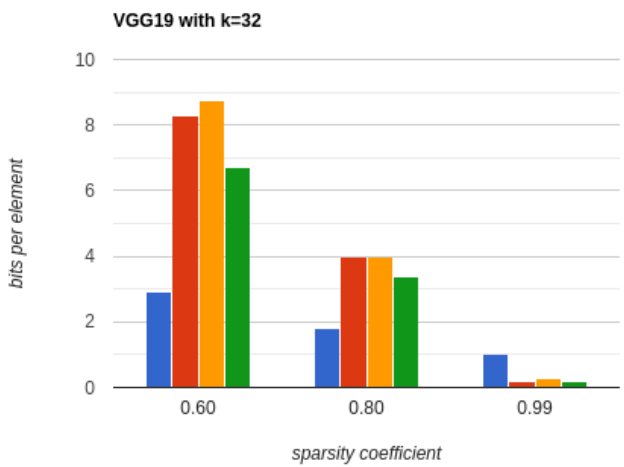
  Fact 2 (worst case): $\psi_{\texttt{sHAM}} \leq s(1 + \log k + \log m) + \frac{\log n}{n} + \frac{B_k}{nm}$,
- **Corollary**: It holds:

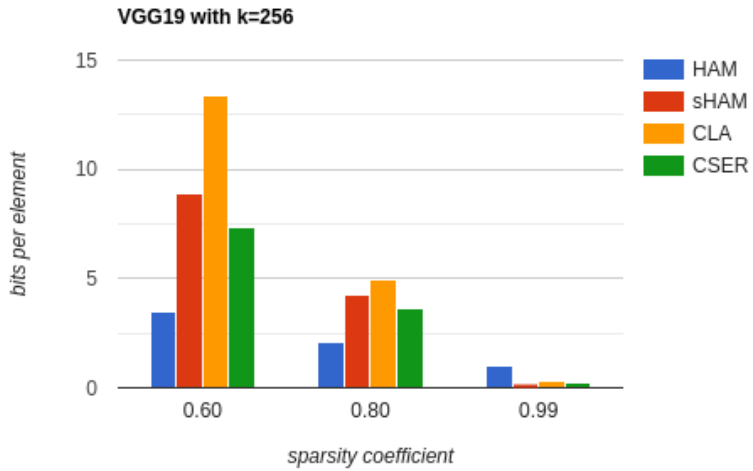
$$s \leq s_{crit} := 1 - \frac{n\log m + \log n}{n(1 + \log k + \log m)} \implies \psi_{\texttt{sHAM}} \leq \psi_{\texttt{HAM}}.$$

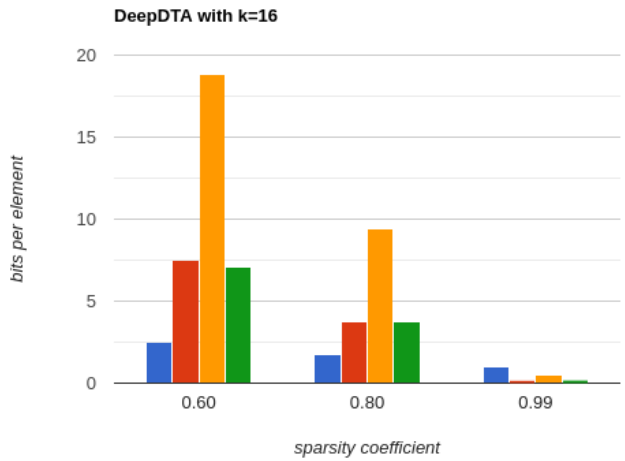
Experimental results

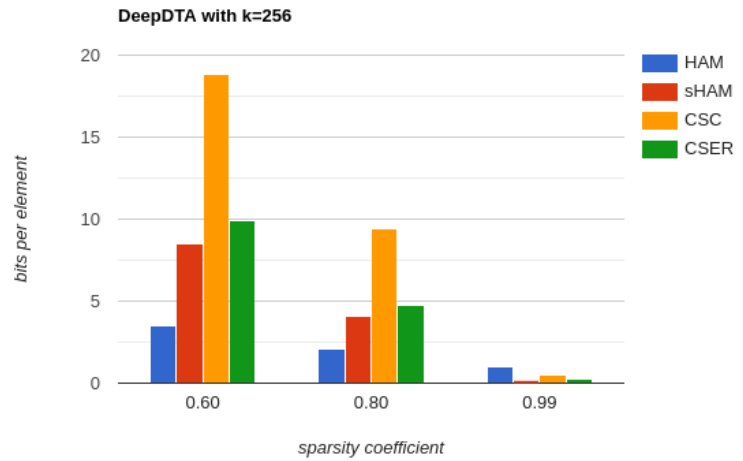
- Dense layers of two publicly available trained networks:
  1. *VGG19* [7], for multiclass classification, trained on MNIST [8] dataset
     - 512×4096
     - 4096×4096
     - 4096×10
  2. *DeepDTA* [9], for predicting proteins and ligands affinity (regression), trained on DAVIS [10]
     - 192×1024
     - 1024×1024
     - 1024×512
     - 512×1
- **Evaluation**: per element storage requirements measured in bits, i.e. the overall structure overhead in bits divided by *nmb*.

Space efficiency









Conclusions

- Given a sparsified and quantized trained DNN/CNN, we have focused on further reducing its memory footprint through sparse formats and source coding
- Several Huffman codes variants have been implemented and tested, keeping an eye on space overhead and decoding time
- Currently, we are working on optimizing time consumption, through parallelization of the SpMV operation.

References I

[1] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398:185–196, 2020.

[2] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, USA, 2nd edition, 2003.

[3] D. J. Rose and R. A. Willoughby, editors. *Sparse Matrices and Their Applications*. New York, NY, USA, 1972.

[4] Ahmed Elgohary et al. Compressed linear algebra for large-scale machine learning. *VLDB J.*, 27(5):719–744, 2018.

[5] Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Compact and computationally efficient representation of deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 31(3):772–785, 2020.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[7] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[8] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Hakime Öztürk et al. DeepDTA: deep drug–target binding affinity prediction. *Bioinformatics*, 34(17):i821–i829, 09 2018.

[10] Mindy I. Davis and other. Comprehensive analysis of kinase inhibitor selectivity. *Nature Biotechnology*, 29:1046–1051, 2011.

[11] Marinò, G., Ghidoli, G., Frasca, M., Malchiodi, D., Compression strategies and space-conscious representations for deep neural networks, *Proceedings of 25th International Conference on Pattern Recognition (ICPR2020)*, 10-15 January, Milan, 2021

[12] Marinò, G., Ghidoli, G., Frasca, M., Malchiodi, D., Reproducing the sparse Huffman Address Map compression for deep neural networks, *Proceedings of 3rd Workshop on Reproducible Research in Pattern Recognition (RRPR2020)*, 11 January, Milan, 2021

Huffman code with a lookup table

The first approach consists of using the original Huffman code, but relying on a **lookup table** for decoding, instead of accessing the tree:

- choose a value t such that $l_{min} \leq t \leq l_{max}$, with l_{min} and l_{max} being the minimum and the maximum code word lengths, respectively
- build a table of size 2^t, indexed by the integer values of the t-sized binary strings
- an entry of the table contains
  - the list of symbols encoded in that substring
  - *rem*, an integer counting the number of bits for which a code word is not available yet

The decoding can be performed by reading the bit stream by chunks of size t, at each step moving forward for t-*rem* positions.

First canonical variant

Starting from an existing Huffman code, to obtain a canonical code we just need to slightly re-arrange the code words and build two arrays:

- *firstcode*, storing the values of the first code words of each length l
- *symb*, storing the list of symbols with code words of each length l

The last entry of *firstcode* is set to 0 and then, down to the first one,

$$firstcode[l] = \lceil (firstcode[l+1] + |symb[l+1]|)/2 \rceil.$$

Example

symbol	*base*	*canonical*
0	0	1
8	100	011
4	1100	0001
5	1111	0010
6	1101	0011
9	1010	0100
10	1110	0101
3	10110	00000
7	10111	00001

length	*firstcode*	*symb*
1	1	0
2	2	
3	3	8
4	1	4, 5, 6, 9, 10
5	0	3, 7

Symbols can be decoded by reading the stream bit by bit, until we obtain a substring of length l with a value $v \geq firstcode[l]$, then the difference v-$firstcode[l]$ gives us the position of $symb[l]$ where the symbol is stored.

Second canonical variant

In this variant we assume symbols are sorted by non-increasing order of their probability. By construction, code words are also assigned so that their values are increasing with respect to their lengths.

The structures required to perform the decoding are:

- *firstsymbol*, an array storing the index of the first symbol assigned to a code word of length l
- *firstcode_l*, an array storing the "left-justified" value, right-padded to l_{max}, of the first code word for each length l
- the list of *symbols* we are encoding

Example

length	*firstsymbol*	*firstcode_l*
0	0	0
1	0	0
2	1	16
3	1	16
4	3	24
5	5	28
6	–	32

symbol	*codeword*
8	011
4	0001
5	0010
6	1101
9	11100
10	11101
3	11110
7	11111

To decode one symbol, we read a chunk of size l_{max} from the stream, searching in $firstcode_l$ for the first element l greater than the integer value of the chunk, then looking at *firstsymbol* to obtain the index of the symbol associated to that code word.

Speed-up table

Searching for the right length l in $firstcode_l$ can be slow:

- adding some space overhead, we can use $2^{l_{max}}$-sized table to reduce the decoding time
- the table is indexed by the integer value of chunks
- an entry contains the length of the first code word for a given chunk.

If we are encoding k symbols, l_max is upper bounded by $k - 1$, making this overhead too heavy when the quantization is not particularly strong.

We replace the table with a partial one of size 2^t, with $l_{min} \leq t < l_{max}$, containing the right length for the first t bits of a chunk, if any, or the first valid l, reducing the time spent searching in $firstcode\_l$.