Compression strategies and space-conscious representations for deep neural networks

Giosuè Cataldo Marinò
Università degli Studi di Milano
Milano, Italy

Gregorio Ghidoli
Università degli Studi di Milano
Milano, Italy

Marco Frasca
Università degli Studi di Milano
Milano, Italy
marco.frasca@unimi.it

Dario Malchiodi
Università degli Studi di Milano
Milano, Italy
dario.malchiodi@unimi.it

***Abstract*—Recent advances in deep learning have made available large, powerful convolutional neural networks (CNN) with state-of-the-art performance in several real-world applications. Unfortunately, these large-sized models have millions of parameters, thus they are not deployable on resource-limited platforms (e.g., where RAM is limited). Compression of CNNs becomes therefore a critical problem to achieve memory-efficient and possibly computationally faster model representations. In this paper, we investigate the impact of lossy compression of CNNs by weight pruning and quantization, and lossless weight matrix representations based on source coding. We tested several combinations of these techniques on four benchmark datasets for classification and regression problems, achieving compression rates up to 165 times, while preserving or improving the model performance.**

***Index Terms*—CNN compression, weight pruning, probabilistic quantization, entropy coding, drug-target prediction**

I. Introduction

Although the main structural results behind deep neural networks (NN) date back to more than forty years ago, this field has constantly evolved, giving rise to original models, but also to novel techniques for controlling generalization error. The availability of powerful computing facilities and of massive amount of data allows nowadays to train extremely efficient neural predictors, setting the state-of-the-art in various fields, such as image processing or financial forecasting. Convolutional neural networks (CNN) play a prominent role: indeed, several pre-trained models, such as AlexNet [1] and VGG16 [2], are made available as a starting point for transfer learning techniques [3]. These models, however, are characterized by a large memory footprint: for instance, VGG16 requires no less than 500 MB to be stored in main memory. As a consequence, querying such models becomes also demanding in terms of energy consumption. This clashes with the limitations of mobile phones, smartwatches, and in general with IoT-enabled devices. Leaving aside the vein of *compact model* production, aiming to directly induce succint CNNs [4], in this paper we focus on *network compression*. Indeed, knowledge in a neural network is *distributed* over millions, or even billions, of connection weights. This knowledge can be extracted transforming a learnt network into a smaller one, yet with comparable (or even better) performance. Main approaches proposed in the literature can be cast into the following four categories:

- *matrix decomposition*, aiming at writing over-informative weight matrices as the product of more compact matrices;
- *data quantization*, focused on limiting the bitwidth of data encoding the mathematical objects behind a NN, such as weights, activations, errors, gradients, and so on;
- *network sparsification*, aimed at reducing the number of free parameters of a NN, notably the connection weights;
- *knowledge distillation*, consisting in subsuming a large network L by a smaller one, trained with the target of mimicking the function learnt by L.

The reader interested to a thorough review of these methods can refer for instance to [5], [6]. Note that we don't consider special techniques for convolutional layers, such as those tweaking or discarding the corresponding filters [7], [8]. This is due to the fact that, in several CNNs, the memory required by these layers is negligible w.r.t. that of fully-connected layers.

This work aims at investigating the joint effect of: (i) lossy compression for NN weights, and (ii) entropy coding, lossless representation techniques aware of the limited amount of memory (aka *space-conscious* techniques [9]). In particular, concerning matrix compression, we analyse the effectiveness of some existing pruning and quantization methods, and introduce a probabilistic quantization technique mutuated from federated learning. From the matrix storage standpoint, we propose a novel representation, named *sparse Huffman Address Map compression* (`sHAM`), combining entropy coding, address maps, and compressed sparse column (CSC) representations. `sHAM` is specifically designed to exploit the sparseness and the quantization of the original weight matrix. The proposed methods have been evaluated on two publicly available CNNs, and on four benchmarks for image classification and for the regression problem of drug-target affinity prediction, confirming that CNN compression can even

improve the performance of uncompressed models, whereas the `sHAM` representation can achieve compression rates up to around 200 times. The work is organized as follows: Sects. II and III describe the considered compression and representation techniques, while Sect. IV illustrates the above mentioned experimental comparison, depicted in terms of performance gain/degradation, achieved compression rate, and execution times. Some concluding remarks end the paper.

II. Compression techniques

This section describes three methodologies used in order to transform the matrix $\boldsymbol{W}^{\circ} \in \mathbb{R}^{n\times m}$ organizing the connection weights of one layer in a neural network into a new matrix $\boldsymbol{W}$ which approximates $\boldsymbol{W}^{\circ}$, though having a structure exploitable by specific compression schemes (cfr. Sect. III) [1]. In the rest of the paper, w° and w will denote generic entries of $\boldsymbol{W}^{\circ}$ and $\boldsymbol{W}$, respectively. Boldface and italic boldface will be used for matrices (e.g., $\boldsymbol{W}$) and vectors ($\boldsymbol{x}$), respectively, while $|\cdot|$ will be an abstract cardinality operator returning the length of a string or the number of elements in a vector. Finally, we define the *sparsity coefficient* $s \in [0,1]$ of $\boldsymbol{W}$ as the ratio between the number of its nonzero elements and mn.

A. Pruning

Neural networks have several analogies with the human central nervous system. In particular, storing knowledge in a distributed fashion implies *robustness* as a side effect: performance degrades gracefully when a damage occurs in the network components, i.e., when connection weights slightly change or get discarded. Robustness can be exploited to compress a NN by removing connections which do not significantly affect the overall behaviour. This is referred to as *pruning* a learnt neural network. An originally oversized NN might even be outperformed by its pruned version.

Pruning is typically done by considering connections whose weights are small in absolute value. Indeed, the signal processed by an activation function is computed as a weighted sum of the inputs to the corresponding neuron, precisely relying on connection weights. Thus, nullifying all negligible (positive or negative) weights should not sensibly change the above signal, as well as the network output. We performed pruning by fixing an empirical percentile w_p of the set of entries of $\boldsymbol{W}^{\circ}$, and subsequently defined the entries of $\boldsymbol{W}$ setting $w = w^{\circ}$ if $|w| > w_p$, 0 otherwise. This procedure has a time complexity of $\mathcal{O}(nm\log(nm))$ (due to sorting). As pruning has the effect of modifying the structure of the neural network, the latter is retrained on the same data, now only updating non-null weights in $\boldsymbol{W}$. The only parameter is the percentile level p, which in turn is obviously related to the sparsity coefficient s (see Sect. IV for a description of how p, as well as k and b in next sections, have been selected).

B. Weight sharing

When the weights in $\boldsymbol{W}^{\circ}$ assume a small number of distinct values, applying the *flyweight* pattern [10] results in a technique called *weight sharing* (WS) [11]. Distinct values are stored in a table, whose indices are used as matrix entries. As (integer) indices require less bits than (float) weights, the latter matrix is significantly compact and largely compensates for the additional table[2]. This comes at the price of requiring two memory accesses in order to retrieve a weight.

Although $\boldsymbol{W}^{\circ}$ isn't expected to initially enjoy this property, robustness is helpful in this case, too. Close weights can be set to a common value without significantly affecting network performance, yet allowing to apply WS. For instance, the approach [11] clusterizes all w° values, setting w to the centroid of the corresponding w°. Assuming the k-means algorithm is used [12], the time complexity is $\mathcal{O}(k(mn)^2)$, where k is the number of different weight values. A second retraining phase is advisable also here, though updating weights is trickier, because the latter shall take values in the centroid set $\{c_1, \ldots, c_k\}$. This is ensured using the cumulative gradient

$$\frac{\partial \mathcal{L}}{\partial c_l} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial w_{ij}} \mathbb{1}(I_{ij} = l),$$

where $l \in \{1, \ldots, k\}$, I_{ij} is the cluster index of w°_{ij}, and $\mathbb{1}$ is the indicator function. Applying cumulative gradient might end up in using less than k distinct weights, if two ore more centroids converge to a same value during retraining. To achieve a higher compression, pruning and WS can be applied in chain, with weight sharing only considering the non-null weights identified by pruning.

C. Probabilistic quantization

A recent trend on quantization relies on probabilistic projections of weight onto special binary [13] or ternary values [14]. Here we present an alternative approach, named *Probabilistic Quantization* (PQ), mutuated and extended from the federated learning context [15], and never used for NN compression. PQ is based on the following probabilistic rationale. Let $\underline{w}$ and $\overline{w}$ denote the minimum and maximum weight in $\boldsymbol{W}^{\circ}$, respectively, and suppose that each learnt weight w° is the specification of a random variable W° distributed according to a fixed, yet unknown, distribution having $\mathcal{W} := [\underline{w}, \overline{w}]$ as support. Let W be the two-valued random variable defined by $\mathrm{P}(W = \underline{w}) = \frac{\overline{w}-w}{\overline{w}-\underline{w}}$ and $\mathrm{P}(W = \overline{w}) = \frac{w-\underline{w}}{\overline{w}-\underline{w}}$. The observations of W approximate a weight w through an extreme form of WS (cfr. previous section), using an approach different from k-means for finding representative weights. Now, the expected value $\mathcal{E}(W|W^{\circ} = w) = w$, and, in turn, $\mathcal{E}(W) = \mathcal{E}(W^{\circ})$ regardless how W° is distributed. Thus, *simulating* W° for each entry w° we obtain an approximation $\boldsymbol{W}$ of $\boldsymbol{W}^{\circ}$ having the desirable *unbiasedness property* that the two corresponding random matrices have the same expected value. This method has been heuristically extended by partitioning $\mathcal{W}$ in $b > 2$ intervals. A generic w° is compressed precisely as in the two-values case, but now $\overline{w}$ and $\underline{w}$ denote now the extremes of the sub-interval containing w°. The sub-intervals can be

[1] these techniques can be applied separately to the dense layers of any NN.

[2] note that, in the original formulation, the representation of this matrix still scales with mn, while in Sect. IV we use a more efficient encoding.

chosen in order to preserve the above mentioned unbiasedness property. This happens when the intervals' extremes are $\chi_{\frac{i}{b}}$, for $i = 1, \ldots, b$, where χ_q denotes the q-quantile of W (this requires the weights to follow a common probability distribution; however, no additional hypotheses are needed). The time complexity of the overall operation is $\mathcal{O}(nm \log(nm))$ (due to quantile computation). Note that the same considerations pointed out for WS at the end of previous section, namely retraining via cumulative gradient formula and combined use with pruning, also apply to PQ.

III. Compressed Matrix Representation

The matrix $\boldsymbol{W}$ obtained using any of the techniques described in Sect. II has as many elements as the original matrix. However, it exhibits properties exploitable by a suitable encoding, so that $\boldsymbol{W}$ is stored using less than $\mathcal{O}(mn)$ memory locations, as required by the classical row-order method. In this section, two existing compressed representations of $\boldsymbol{W}$ are first described, then a novel method is proposed, overcoming their limitations and explicitly profiting from sparsity and repeated values. Moreover, the method does not require assumptions on the matrix sparsity, on the distribution of nonzero elements, or on the presence of repeated values. To be able to efficiently compute the dot product $\boldsymbol{x}^T\boldsymbol{W}$, where $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$, necessary for the forward computation in a NN, a dedicated procedure is also described.

A. Compressed sparse column

The *compressed sparse column* (CSC) format [16] is a common general storage format for sparse matrices. It is composed of three arrays:

- $\boldsymbol{nz}$, containing the nonzero values, listed by columns;
- $\boldsymbol{ri}$, containing the row indices of elements in $\boldsymbol{nz}$;
- $\boldsymbol{cb}$, where the difference $cb_{i+1} - cb_i$ provides the number of nonzero elements in column i; thus, $\boldsymbol{cb}$ has dimension $m+1$, where $cb_{m+1} = cb_1 + |\boldsymbol{nz}|$.

As an example, consider the matrix

$$\boldsymbol{W} = \begin{pmatrix} 1 & 0 & 4 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{pmatrix}, \tag{1}$$

whose corresponding CSC representation is $\boldsymbol{nz} = \begin{pmatrix}1 & 2 & 10 & 3 & 4 & 5 & 6\end{pmatrix}$, $\boldsymbol{ri} = \begin{pmatrix}1 & 3 & 2 & 3 & 1 & 3 & 5\end{pmatrix}$, and $\boldsymbol{cb} = \begin{pmatrix}1 & 3 & 5 & 6 & 6 & 8\end{pmatrix}$. Let $q = |\boldsymbol{nz}|$ be the number of nonzero elements in $\boldsymbol{W}$, and B be the number of bits used to represent every element of the matrix (one memory word), so that we need Bnm bits to store $\boldsymbol{W}$, and $(2q+m+1)B$ to store its CSC representation. Note that we assumed B bits are needed also for the components of $\boldsymbol{ri}$, although they can be represented using only $\lceil \log n \rceil$ bits, which might be lower than B. Thus the occupancy proportion is given by $\psi_{CSC} = \frac{2q+m+1}{nm}$.

Denoting by s the sparsity coefficient of $\boldsymbol{W}$, we have $q = snm$, thus $\psi_{CSC} < 1$ implies $s < \frac{1}{2}$. The matrix dot product $\boldsymbol{x}^T\boldsymbol{W}$ will be computed through the typical dot product for CSC format, with computational complexity $\mathcal{O}(q)$ [16], that can be sped up through parallel computing. The main limitation of CSC is the use of B bits for any elements of the matrix, whereas variable length coding can provide more compact representations and higher bit-memory efficiency.

B. Huffman address map compression

The idea of using Huffman coding after network pruning and quantization is introduced in [11], although not in detail. Here, in addition to provide an exhaustive description, we also point out a main limitation of this approach: it does not directly profit from the matrix sparsity. Like CSC, this is a lossless compression technique, based on Huffman coding and the address map logic [17], which we named *Huffman Address Map compression* (HAM). In address maps, matrix elements are treated as a sequence of bits, concatenated by rows or by columns, where null entries correspond to the bit 0, and each nonzero element z is substituted by a binary string encoding its address $a(z)$, and concatenated to the rest of the stream. For instance, the bit stream for the matrix defined in (1) is

$$a(1)0a(2)000a(10)a(3)00a(4)00000000000a(5)0a(6)\ .$$

To be efficient, this storage needs a compact representation of addresses. The Huffman coding $H_{\boldsymbol{W}}(z)$ of nonzero values z is a uniquely decodable and instantaneous code ensuring a near-optimal compression rate [18]. Indeed, given a source $(w_1, \ldots, w_l)$ whose symbols have probabilities $(p_1, \ldots, p_l)$, the average number of bits per symbol $\overline{H}_{\boldsymbol{W}} := \sum_{i=1}^{l} p_i |H_{\boldsymbol{W}}(w_i)|$ is almost equal to the optimal value $\mathcal{H} = -\sum_{i=1}^{l} p_i \log p_i$ corresponding to the entropy of the source (when the symbols are independent and identically distributed). More precisely, $\mathcal{H} \leq |\overline{H}_{\boldsymbol{W}}| \leq \mathcal{H} + 1$, and $\mathcal{H}$ corresponds to the minimal average number of bits per symbol, according to Shannon's source coding theorem [19].

Once the Huffman code $H_{\boldsymbol{W}}$ has been built, we replace each $a(z)$ in the bit stream with the corresponding $H_{\boldsymbol{W}}(z)$. In order to have a uniquely decodable string, zeros are also included in the Huffman code, thus having a total of $q+1$ codewords. The resulting bit stream $\mathtt{HAM}(\boldsymbol{W})$ is then split into $N = \lceil \frac{|\mathtt{HAM}(\boldsymbol{W})|}{B} \rceil$ memory words, $\mathtt{HAM}(\boldsymbol{W})_1, \ldots, \mathtt{HAM}(\boldsymbol{W})_N$, represented as an array $\mathcal{C}_{\mathtt{HAM}}(\boldsymbol{W})$ of N unsigned integers. If $|\mathtt{HAM}(\boldsymbol{W})|$ is not a multiple of B, zero-padding is added to the last word.

To estimate $|\mathtt{HAM}(\boldsymbol{W})|$, we can assume the worst case for the entropy value, that is when all symbols are distinct (nm symbols appearing exactly once in the matrix): in this case $\mathcal{H} = \log(nm)$, and the Huffman code has an average codeword length upper-bounded by $1 + \log(nm)$, which implies at most $nm(1 + \log(nm))$ bits are needed. To store $H_{\boldsymbol{W}}$, and its inverse $H_{\boldsymbol{W}}^{-1}$ used to decode, we need extra space: although there are methods storing a n-symbols Huffman code using at most $\lceil 10.75n \rceil - 3$ bits [20], to ensure optimal search time, a 'classical' B-tree representation is used for both $H_{\boldsymbol{W}}$ and $H_{\boldsymbol{W}}^{-1}$, being aware that space occupancy can be improved. Assuming each value is represented trough 1 word (B bits), each dictionary requires $3(q+1)B$ bits, $2B$ to store each

pair z and $H(z)$, and B bits to store a pointer in the B-tree structure—overestimated, since we have less pointers than keys in a B-tree. Overall, HAM requirements are upper-bounded by $nm(1+\log(nm))+6mnB$ bits, which is more than mnB bits required by an uncompressed matrix. For this reason, in the experiments only using pruning, the CSC representation is adopted. As opposite, the space occupancy of HAM decreases when only k distinct weights are present in $\boldsymbol{W}$, like in the output of WS and PQ (see Sections II). Indeed, in the worst case (all symbols are equally probable, entropy $\log k$), the occupancy proportion is at most $\psi_{\texttt{HAM}} = \frac{1+\log k}{B} + \frac{6k}{nm}$, where as expected, for small k the first term is more relevant, while the second term grows faster with k.

Dot product. The procedure $\texttt{Dot}_{\texttt{HAM}}$ (Fig. 1) executes the dot product $\boldsymbol{x}^T\boldsymbol{W}$, when $\boldsymbol{W}$ is represented through the HAM format. It processes one compressed word of $\mathcal{C}_{\texttt{HAM}}(\boldsymbol{W})$ at a time, obtains its binary representation S (line 3), which is scanned at lines 4-10 to detect code words. The procedure NCW gets the next code word from S, starting at the current bitstring offset $oset$, possibly adding at the beginning of S the bits rem remaining from previous word. If, starting from the current offset, no code word can be detected in S (NCW returns $null$), it means that the next code word has been split on two adjacent memory words, accordingly rem is updated and the next word will be read in the next iteration at line 2. The procedure also takes into account for the 0 padding. Then, the weight relative to the code word detected is computed (line 5), and multiplied by the corresponding element of $\boldsymbol{x}$, to update the cumulative sum stored in variable sum, thus requiring to keep in memory only one weight at a time.

In summary, the N iterations take time $\mathcal{O}(NB) = \mathcal{O}(nm)$ for line 3, $\mathcal{O}(N)$ for lines 5-8, and $\mathcal{O}(nm \log k)$ for line 4, leading to an overall time complexity $\mathcal{O}(nm \log k)$.

Fig. 1. Pseudocode of the dot procedure for HAM representation.

Procedure $\texttt{Dot}_{\texttt{HAM}}$
Input: compressed array $\mathcal{C}_{\texttt{HAM}}(\boldsymbol{W})$; decoding dictionary $H^{-1}_{\boldsymbol{W}}$; vector $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$; number of compressed words N;
begin algorithm
01: Initialize: out :=zeros(n), $row := 1$, $col := 1$
$sum := 0$, $rem := null$, $oset := 0$
02: **for each** i **from** 1 **to** N **do**
03: $S :=$ getBinarySeq($\mathcal{C}_{\texttt{HAM}}(\boldsymbol{W})[i]$)
04: **while** $[oset, rem, z] :=$ NCW($S, rem, oset$) $\neq null$ **do**
05: $sum := sum + x[row] * H^{-1}_{\boldsymbol{W}}(z),\ row := row + 1$
06: **if** $row > n$ **then**
07: $row := 1,\ out[col] := sum$
08: $col := col + 1,\ sum := 0$
09: **end if**
10: **end while**
11: **end for**
end algorithm
Output: out, that is $\boldsymbol{x}^T\boldsymbol{W}$.

C. *Sparse Huffman address map compression*

One drawback of HAM representation is that it does not directly exploit the sparsity of the matrix, which only indirectly induces a reduction in the space occupancy, due to the more compact resulting Huffman code (symbol 0 has high frequency). When the matrix is large and very sparse, even using only 1 bit to represent the symbol 0, much memory would be required (e.g., $10s$ GB for a $10^5 \times 10^5$ matrix). To address this issue, the novel *sparse Huffman Address Map compression* (sHAM) is proposed, extending the HAM format as follows. The symbol 0 is excluded from the bit stream and from the Huffman code, and a bitwise CSC representation of the matrix is adopted, producing the vectors $\boldsymbol{nz}, \boldsymbol{ri}, \boldsymbol{cb}$ (cfr. Sect. III-A), but storing $\boldsymbol{nz}$ using the HAM format. Namely, the Huffman code $H_{\boldsymbol{nz}}$ for nonzero elements is built, and the corresponding bit stream $\texttt{sHAM}(\boldsymbol{nz}) = H_{\boldsymbol{nz}}(nz[1]) \dots H_{\boldsymbol{nz}}(nz[q])$ is obtained by concatenating their Huffman coding. $\texttt{sHAM}(\boldsymbol{nz})$ is stored in the array $\mathcal{C}_{\texttt{sHAM}}(\boldsymbol{nz})$ of $N_1 = \lceil \frac{|\texttt{sHAM}(\boldsymbol{nz})|}{B} \rceil$ memory words. Considering the worst case, in which all $q = snm$ symbols are distinct, $\mathcal{C}_{\texttt{sHAM}}(\boldsymbol{nz})$ is composed of $snm(1 + \log snm)$ bits, in addition to the $6snmB$ for the dictionaries, and to the $B(n + m + 1)$ bits required for $\boldsymbol{ri}$ and $\boldsymbol{cb}$. The resulting occupancy ratio is $\psi_{\texttt{sHAM}} = \frac{s(1+\log snm)}{B} + 6s + \frac{n+m+1}{nm}$.

On the other side, when only k distinct values are present in $\boldsymbol{nz}$, and again assuming the worst case for the Huffman coding, the occupancy ratio becomes $\psi_{\texttt{sHAM}} = \frac{s(1+\log k)}{B} + \frac{6k}{nm} + \frac{n+m+1}{nm}$, where first term on the right is scaled by s w.r.t. $\psi_{\texttt{HAM}}$, emphasizing the gain of increasing the sparsity of $\boldsymbol{W}$, whereas last term is constant w.r.t. k and s. Thus, when s is such that $\frac{s(1+\log k)}{B} + \frac{n+m+1}{nm} < \frac{(1+\log k)}{B}$, it follows $\psi_{\texttt{sHAM}} < \psi_{\texttt{HAM}}$.

Dot product. Figure 2 describes the procedure executing the dot product $\boldsymbol{x}^T\boldsymbol{W}$ when $\boldsymbol{W}$ is represented through the sHAM format. It extracts in sequence the compressed words of $\mathcal{C}_{\texttt{sHAM}}(\boldsymbol{nz})$, computes the corresponding binary representation S (line 3), and executes lines 4-13 to detect code words. NCW is the same procedure used for $\texttt{Dot}_{\texttt{sHAM}}$, whereas the cycle at lines 5-7 possibly skips empty columns. The variable pos contains the position in $\boldsymbol{nz}$ of the current element z. Line 8 finds the weight relative to the code word detected, multiplies it by the corresponding element of $\boldsymbol{x}$, and updates the column cumulative sum stored in variable sum. The N_1 iterations require $\mathcal{O}(N_1 B) = \mathcal{O}(snm)$ steps for line 3, $\mathcal{O}(N_1)$ for lines 8-12, $\mathcal{O}(m)$ for cycle 5-7, and $\mathcal{O}(snm \log k)$ for line 4. Thus, the overall time complexity is $\mathcal{O}(snm \log k)$.

The procedure $\texttt{Dot}_{\texttt{sHAM}}$ can be adapted to parallel computation by substituting $\boldsymbol{cb}$ with the vector containing the beginning of each column in the bitstream $\texttt{sHAM}(\boldsymbol{nz})$, and using the current position in the bitstream ($oset$) to detect the end of columns. In this way, each column product can be run in parallel (e.g., through GPU); we considered this as a future development, since in the empirical evaluation the sequential version was very close to the full matrix parallel dot product testing time, on sufficiently sparse and quantized matrices. Analogously, also $\texttt{Dot}_{\texttt{HAM}}$ can be parallelized.

Fig. 2. Pseudocode of the dot procedure for `sHAM` representation.

Procedure $\texttt{Dot}_{\texttt{sHAM}}$
`Input`: compressed array $\mathcal{C}_{\texttt{sHAM}}(\boldsymbol{nz})$; row index vector $\boldsymbol{ri}$; vector $\boldsymbol{cb}$; vector $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$; decoding dictionary $H_{\boldsymbol{nz}}^{-1}$; number of compressed words N_1;
`begin algorithm`
01: Initialize: out :=zeros(n), $pos := 1$, $col := 1$
$sum := 0$, $rem := null$, $oset := 0$
02: **for each** i **from** 1 **to** N_1 **do**
03: $S :=$ getBinarySeq($\mathcal{C}_{\texttt{sHAM}}(\boldsymbol{nz})[i]$)
04: **while** $[rem, oset, z] :=$ NCW$(S, rem, oset) \neq null$ **do**
05: **while** $cb[col+1] = pos$ **do**
06: $col := col + 1$, $out[col] := 0$
07: **end while**
08: $sum := sum + x[ri[pos]] * H_{\boldsymbol{nz}}^{-1}(z)$, $pos := pos + 1$
09: **if** $cb[col+1] = pos$ **then**
10: $out[col] := sum$
11: $sum := 0,\ col := col + 1$
12: **end if**
13: **end while**
14: **end for**
`end algorithm`
`Output`: out, that is $\boldsymbol{x}^T\boldsymbol{W}$.

IV. Experiments and Results

To assess the quality of the proposed techniques, an empirical evaluation has been carried out on four datasets and two uncompressed models, as explained here below.

A. Data

- *Classification*. The MNIST database [21] is a classical large database of handwritten digits, containing 60K+10K 28x28 grayscale images (train and test set, respectively) from 10 classes (digits 0-9). The CIFAR-10 dataset [22] consists of 50K+10K 32x32 color images belonging to 10 different classes. Both datasets are balanced w.r.t. labels.
- *Regression*. We predicted the affinity between drug (ligand) and targets (proteins) [23], using the DAVIS [24] and KIBA [25] datasets. Proteins and ligands are both represented through strings, respectively using the amino acid sequence and the SMILES (Simplified Molecular Input Line Entry System) representation. DAVIS and KIBA contain, respectively, 442 and 229 proteins, 68 and 2111 ligands, 30056 and 118254 total interactions.

B. Benchmark models

To have a fair comparison of the various compression techniques, we selected top-performing CNN models publicly available: (i) *VGG19* [2], made up by 16 convolutional layers and a fully-connected block (two hidden layers of 4096 neurons each, and a softmax output layer)[3], trained on CIFAR-10 and MNIST datasets; and (ii) *DeepDTA* [23], with distinct convolutional blocks for proteins and ligands (each composed of 3 convolutional and a MaxPool layers), combined in a fully connected block consisting of 3 hidden layers of 1024, 1024, 512 units, and a single-neuron output layer[4].

The original work using DeepDTA operated a 5-fold cross validation (CV) to perform model selection, thus training on 4/5 of available data. We retained the best configuration for hyperparameters and trained the CNN on the entire training set, leaving unchanged the original settings.

C. Evaluation metrics

We considered the difference $\Delta_{\rm perf}$ between performances of compressed and uncompressed models, the ratio of testing time of the uncompressed model w.r.t. the compressed one (named $time$), and the space occupancy ratio ψ (cfr. Sect. III-A). As in the original works, we computed performance using *Accuracy* and *mean squared error* (MSE) for classification and regression, respectively. Time and space performance account only for the actually compressed weights, that is those in fully-connected layers. Moreover, the implementation of dot product for non-compressed models exploits parallel computations implemented in Python, thus penalizing `HAM` and `sHAM`, implemented sequentially (their parallelization is planned as an extension). As shown in the next section, even with this penalty, our method approaches the uncompressed time when the matrix is sufficiently sparse and quantized.

D. Compression techniques setup

We tested all combinations of Pruning (Pr), WS, PQ, Pr-WS, Pr-PQ, selecting hyper-parameters as follows.

- *Pruning*. We tested percentiles with $p \in \{30, 40, 50, 60, 70, 80, 90, 95, 96, 97, 98, 99\}$; values 30 and 40 (for which CSC does not achieve any compression) are included, as potentially useful in Pr-WS and Pr-PQ.
- *WS*. For VGG19, $k = 2, 32, 128, 1024$ was tested in the first two hidden layers, and $k = 2, 32$ in the third one (which is smaller). DeepDTA is smaller than VGG19, so we retained the combinations $k = 2, 32, 128$ in the two hidden layers, and $k = 2, 32$ in the output layer.
- *PQ*. In order to have a fair comparison, b took the same values as k in the WS procedure;
- *Pr-X*. The combined application of pruning followed by the quantization $X \in \{\mathrm{WS}, \mathrm{PQ}\}$, was tested in two variants: a) best p in terms of $\Delta_{\rm perf}$ is selected, and the parameters for X are subsequently tuned a in previous points; b) the vice-versa.

Fine tuning of compressed weights. The same configuration of original training procedure has been kept for the retraining after compression. Data-based tuning was applied only to learning rate after retraining ($3 \cdot 10^{-4}$ for pruning, 10^{-3} and 10^{-4} for PQ, WS, and combined schemes), and the maximum number of epochs, set to 100.

E. Software implementation

The source code retrieved for baseline NNs was implemented in Python, using the Tensorflow and Keras libraries.

[3]Source code: `https://github.com/BIGBALLON/cifar-10-cnn`

[4]Source code: `https://github.com/hkmztrk/DeepDTA`

TABLE I

TESTING PERFORMANCE OF ORIGINAL NON-COMPRESSED MODELS. PERFORMANCE SHOWS ACCURACY FOR MNIST AND CIFAR-10, AND MSE FOR KIBA AND DAVIS. *Time* IS THE OVERALL TESTING TIME.

Net	Dataset	Performance	Time (s)
VGG19	MNIST	0.9954	$8.88 \cdot 10^{-1}$
	CIFAR10	0.9344	$8.97 \cdot 10^{-1}$
DeepDTA	KIBA	0.1756	$1.75 \cdot 10^{-1}$
	DAVIS	0.3223	$4.00 \cdot 10^{-2}$

Compression techniques and retraining procedures have been implemented in Python as well, also exploting GPUs[5].

F. Results

As baseline comparison, Table I reports the testing results of the uncompressed models. The top performing results for each compression technique, along with the corresponding configuration, are shown in Table II. To also evaluate compression capability, Table III contains the least occupying configuration for each compression methods having performance greater or equal to the original model (when available). Weight quantization is more accurate than pruning for classification, with PQ and WS having the top absolute performance on MNIST and CIFAR-10, respectively. Overall, all techniques outperform the baseline, while exhibiting remarkable compression rates. Similar trends raise for regression, where however pruning top-performs, and where PQ never improves the baseline MSE on KIBA dataset. Performance improvements are particularly remarkable on DAVIS data (till around 30% of baseline). As expectable, the largest compression (while preserving accuracy) is achieved on the bigger net, VGG19, with a compression rate of more than 150 times on CIFAR-10, with Pr-PQ and `sHAM` representation. Anyway, Pr-PQ method improves the baseline MSE of 17.1%, while compressing around 18 times, also for DeepDTA.

To better unveil the behavior of the proposed compression methods and storage formats, in Fig. 3 we summarize their space occupancy, time ratio, and testing performance for all tested hyper-parameter configurations. The `sHAM` storage format is used, except for techniques producing dense matrices, where `HAM` is more convenient. Reminding that WS and PQ combinations are reported in increasing order (before all combinations with $k, b = 2$ in the first layer, denoted by label 2, then those with $k, b = 32$ in the first layer, label 32, and so on), on CIFAR-10 and DAVIS data most compression techniques outperform the baseline, and this is likely due to overfitting, since on training data they show similar results. Conversely, on MNIST and KIBA only some compression configurations improve the baseline results, which is however important, since at the same time the compressed model uses much less parameters, confirming results obtained in [11]. When using binary quantization ($k, b = 2$) clearly we get lower ψ but at the same time worse performance, whereas already

[5]Source code, datasets and trained baseline networks are available at `https://github.com/giosumarin/ICPR2020_sHAM`.

TABLE II

TOP TESTING PERFORMANCE ACHIEVED BY COMPRESSION TECHNIQUES. *Type* IS THE COMPRESSION TECHNIQUE, WHILE *Perf* CONTAINS ACCURACY FOR VGG19 AND MSE FOR DEEPDTA. ψ IS THE OCCUPANCY RATIO, WHEREAS $*$ DENOTES `sHAM` REPRESENTATION AS THE LOWEST OCCUPANCY ON THAT SETTING (W.R.T. `HAM`). IN BOLD THE BEST RESULTS ON EACH COUPLE NET-DATASET.

Net-Dataset	Type	Configuration	Perf	ψ
VGG19-MNIST	Pr	96	0.9954	0.08
	WS	128-32-32	0.9957	0.321
	PQ	32-32-2	**0.9958**	0.309
	Pr-WS a	96/128-32-32	0.9956	0.039*
	Pr-WS a	96/128-32-32	0.9956	0.039*
	Pr-PQ a	96/32-128-32	0.9956	**0.026***
	Pr-PQ b	50/32-32-2	**0.9958**	0.187
VGG19-CIFAR10	Pr	60	0.9365	0.8
	WS	32-32-2	**0.9371**	0.306
	PQ	32-2-32	0.9363	0.091
	Pr-WS a	60/2-2-32	0.9366	0.088
	Pr-WS b	50/32-32-2	0.9370	0.216
	Pr-PQ a	60/2-2-32	0.9363	0.088
	Pr-PQ b	98/32-2-32	0.9365	**0.012***
DeepDTA-KIBA	Pr	60	**0.1599**	0.8
	WS	128-128-32-2	0.1679	0.390
	PQ	32-128-128-32	0.1761	0.425
	Pr-WS a	60/32-128-2-32	0.1666	**0.187**
	Pr-WS b	30/128-128-32-2	0.1644	0.33
	Pr-PQ a	60/128-128-128-32	0.1769	0.207
	Pr-PQ b	40/32-128-128-32	0.1683	0.291
DeepDTA-DAVIS	P	80	**0.2242**	0.4
	WS	128-2-128-2	0.2320	0.212
	PQ	128-32-32-32	0.2430	0.324
	Pr-WS a	80/32-128-2-32	0.2341	**0.105**
	Pr-WS b	40/128-2-128-2	0.2826	0.191
	Pr-PQ a	80/128-128-32	0.2302	0.122
	Pr-PQ b	60/128-32-32-32	0.2353	0.160

TABLE III

BEST OCCUPANCY RATIO ENSURING NO DECAY IN PERFORMANCE W.R.T. UNCOMPRESSED MODEL. SAME NOTATIONS AS IN TABLE II.

Net-Dataset	Type	Configuration	Perf	ψ
VGG19-MNIST	Pr	97	0.9953	0.06
	WS	128-2-32	0.9954	0.104
	PQ	1024-2-32	0.9955	0.126
	Pr-WS a	96/2-128-2	0.9954	0.038*
	Pr-WS b	96/128-32-32	**0.9956**	0.039*
	Pr-PQ a	96/32-128-32	**0.9956**	0.026*
	Pr-PQ b	97/32-32-2	0.9955	**0.018***
VGG19-CIFAR10	Pr	99	0.9357	0.02
	WS	2-2-32	0.9360	0.063
	PQ	2-2-32	0.9351	0.063
	Pr-WS a	60/2-2-32	**0.9366**	0.088
	Pr-WS b	99/32-32-2	0.9358	**0.006***
	Pr-PQ a	60/2-2-32	0.9363	0.088
	Pr-PQ b	99/32-2-32	0.9363	**0.006***
DeepDTA-KIBA	Pr	60	**0.1599**	0.8
	WS	32-32-2-2	0.1723	0.228
	PQ	32-128-128-32	0.1761	0.425
	Pr-WS a	60/32-2-32-2	0.1739	**0.127**
	Pr-WS b	60/128-128-32-2	0.1712	0.222
	Pr-PQ a	60/128-128-128-32	0.1769	0.207
	Pr-PQ b	50/32-128-128-32	0.1702	0.243
DeepDTA-DAVIS	Pr	90	0.2425	0.2
	WS	2-2-2-2	0.2840	0.063
	PQ	32-32-2-32	0.2567	0.237
	Pr-WS a	80/32-2-2-32	**0.2367**	0.079
	Pr-WS b	60/128-2-128-2	0.2906	0.148
	Pr-PQ a	90/128-32-32-32	0.2671	**0.060***
	Pr-PQ b	80/32-2-2-32	0.2943	0.077

with $k, b = 32$ the baseline performance is improved on almost all datasets. sHAM occupancy, as expected, gets lower when p increases (and consequently s decreases), along with the time ratio, approaching in turn to 1 (same testing time). The high time ratios, for some configurations, reflect the fact that the compress dot procedure is slower than the numpy.dot used by baseline and leveraging parallel computation. As mentioned in Sect. IV-C, the former is still sequential, and we plan to produce a parallel version (cfr. Sect. III-C).

Comprehensively, a compression technique better that the other ones is not emerging from these results. Nevertheless, a sound result is that methods providing the lowest occupancy, i.e., those combining weight pruning and quantization, still achieve high performance; secondly, it seems that the pruning technique is preferable for regression problems, whereas quantization performs better in the setting of classification. However, we believe further studies are necessary to assume this trend as consolidated. Finally, our proposed storage representations, HAM and sHAM, produce the expected behaviors, being suitable for both dense (HAM) and sparse (sHAM) compressed matrices, and remarkably improving the CSC format on sparse matrices.

V. Conclusions

This work investigated both classical CNN compression techniques (like weight pruning and quantization) and a novel probabilistic compression algorithm, combined with a new lossless entropy coding storage of the network. Our results confirmed that model simplification can improve the overall generalization abilities, e.g., due to limited overfitting, and showed that our compressed representation can reduce the space occupancy of the input network, when suitably preceded by pruning and quantization, more than 150 times. As meaningful extension of this work, it would be worthy to include other quantization techniques (e.g. the incremental quantization [26]), and to operate the quantization so as to minimize the entropy of the quantized weights (known as entropy coded scalar/vector quantization), which in turn would lead to shorter entropy coding [27]. In this study indeed we considered them separately, since the aim was to compare the effectiveness in terms of prediction accuracy of different compression techniques, to detect possible performance trends related to the type of problem. Moreover, other source coding methodologies (known as universal lossless source coding, e.g., the Lempel–Ziv source coding), less sensitive to source statistics, could be applied rather than Huffman coding, being more convenient in practice than the latter, since they do not require the knowledge of source statistics, and having smaller overhead, since the codebook (i.e., dictionary) is built from source symbols while encoding and decoding.

Acknowledgements

This work has been supported by the Italian MUR PRIN project "Multicriteria data structures and algorithms: from compressed to learned indexes, and beyond" (Prot. 2017WR7SHH).

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[3] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[4] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[5] L. Deng *et al.*, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[6] Y. Cheng *et al.*, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.

[7] S. Zhai *et al.*, "Doubly convolutional neural networks," in *Advances in neural information processing systems*, 2016, pp. 1082–1090.

[8] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5068–5076.

[9] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.

[10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR 2016*, 2015, arxiv:1510.00149.

[12] J. B. McQueen, "Some methods of classification and analysis in multivariate observations," in *Proc. of fifth Barkley symposium on mathematical statistics and probability*, 1967, pp. 281–297.

[13] M. Courbariaux *et al.*, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, 2015, pp. 3123–3131.

[14] L. Deng *et al.*, "Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018.

[15] J. Konečný *et al.*, "Federated learning: Strategies for improving communication efficiency," in *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. [Online]. Available: https://arxiv.org/abs/1610.05492

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

[17] U. W. Pooch and A. Nieder, "A survey of indexing techniques for sparse matrices," *ACM Comput. Surv.*, vol. 5, no. 2, pp. 109–133, Jun. 1973. [Online]. Available: https://doi.org/10.1145/356616.356618

[18] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[19] C. E. Shannon, "A mathematical theory of communication." *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, 1948.

[20] Z. Sultana and S. Akter, "A new approach of a memory efficient huffman tree representation technique," in *2012 International Conference on Informatics, Electronics Vision (ICIEV)*, 2012, pp. 731–736.

[21] 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, 1998.

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, University of Toronto, 2009.

[23] H. Öztürk *et al.*, "DeepDTA: deep drug–target binding affinity prediction," *Bioinformatics*, vol. 34, no. 17, pp. i821–i829, 09 2018.

[24] M. I. Davis and other, "Comprehensive analysis of kinase inhibitor selectivity," *Nature Biotechnology*, vol. 29, pp. 1046–1051, 2011.

[25] J. Tang *et al.*, "Making sense of large-scale kinase inhibitor bioactivity data sets: A comparative and integrative analysis," *Journal of Chemical Information and Modeling*, vol. 54, no. 3, pp. 735–743, 2014.

[26] A. Zhou *et al.*, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017. [Online]. Available: http://arxiv.org/abs/1702.03044

[27] Y. Choi *et al.*, "Universal deep neural network compression," *IEEE Journal of Selected Topics in Signal Processing*, pp. 1–1, 2020, in press.

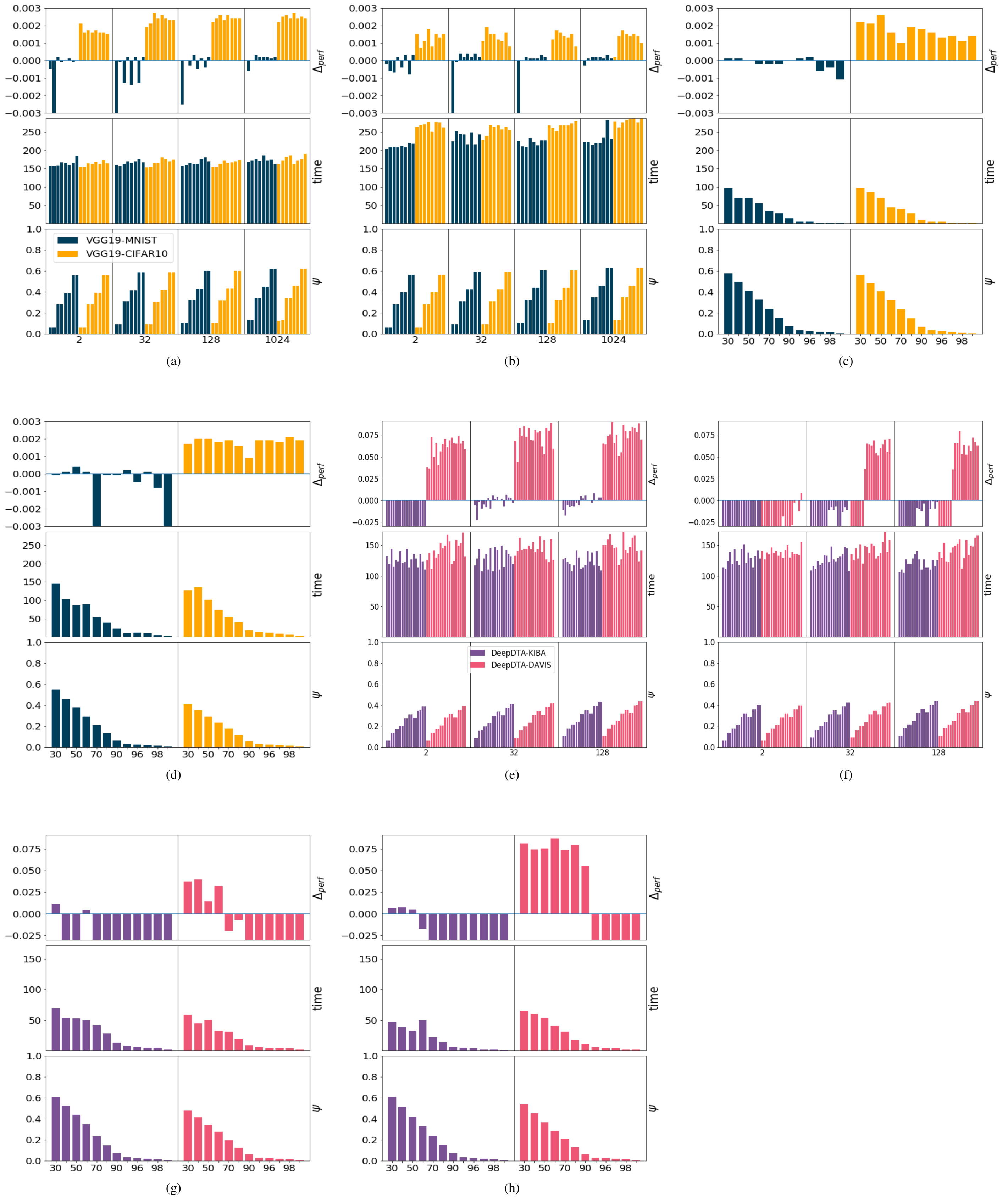

Fig. 3. Overall testing performance for compression methods: a) WS (`HAM` format), b) PQ (`HAM`), c) Pr-WS a (`sHAM`), d) Pr-PQ a (`sHAM`), e), f), g) and h) the same a), b), c) and d) methods, but on regression datasets. X-axis report the compression parameters, while Y-axis measures the performance in term of space occupancy, time ratio, and testing performance (from top to bottom).