IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 28, 2020, accepted November 4, 2020, date of publication November 10, 2020,
date of current version November 20, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3037254

Hardware-Based Real-Time Deep Neural Network Lossless Weights Compression

TOMER MALACH[1], SHLOMO GREENBERG[1], (Member, IEEE), AND MOSHE HAIUT[2]
[1]School of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Be'er Sheva 84105, Israel
[2]DSP Group Inc., Herzliya 4659071, Israel

Corresponding author: Shlomo Greenberg (shlomog@bgu.ac.il)

This work was supported in part by the GenPro Consortium within the Framework of the Israel Innovation Authority's MAGNET Program.

ABSTRACT Deep Neural Networks (DNN) are widely applied to many mobile applications demanding real-time implementation and large memory space. Therefore, it presents a new challenge for low-power and efficient implementation of a diversity of applications, such as speech recognition and image classification, for embedded edge devices. This work presents a hardware-based DNN compression approach to address the limited memory resources in edge devices. We propose a new entropy-based compression algorithm for encoding DNN weights, as well as a real-time decoding method and efficient dedicated hardware implementation. The proposed approach enables a significant reduction of the required DNN weights memory (approximately 70% and 63% for AlexNet and VGG19, respectively), while allowing the decoding of one weight per clock cycle. Results show a high compression ratio compared to well-known lossless compression algorithms. The proposed hardware decoder enables an efficient implementation of large DNN networks in low-power edge devices with limited memory resources.

INDEX TERMS Deep neural network, entropy compression, hardware decoder, real-time.

I. INTRODUCTION

Deep Neural Networks (DNNs) have become a powerful tool for many Artificial Intelligence (AI) applications such as computer vision, robotics, and NLP. Various types of DNN architectures have recently been proposed, such as: AlexNet, GoogLeNet, VGG, and ResNet. Ever since AlexNet [1] won the ILSVRC2012 image classification challenge (demonstrating 16.4% top-5 error rate for the ImageNet 2012 data-set) the field of Neural Networks (NNs) has gained significant momentum. During the last decade, more complex DNNs have been developed showing outstanding performances and continuous improvement accuracy [2]–[5], [6]. Russakovsky *et al.* [7] evaluate human accuracy in image classification on the ILSVRC 2012 data-set, and conclude that humans outperform GoogLeNet only by approximately 1.7% (6.8% vs. 5.1% top-5 error rate, respectively). However, subsequently the ResNet [4] network achieved 3.57% top-5 error rate on the same data set, outperforming humans.

DNN architectures are composed of several NN layers and are considered deep networks that are tightly connected. Typically, the input data connected to the first layer is processed and transferred to the next layer and so on, until all data reaches the output layer. DNN can consist of many different types of layers, such as: Fully Connected (FC), Convolution (Conv), Pooling, Dropout, Batch Normalization and others. State-of-the-art DNNs include multiple layers and require a substantial amount of memory for storing the weights. Some of the NN layers, such as FC or Conv, require a large memory space. For example, an FC layer with an input structure of 256×256x3 and 128 neurons contains about 24M weights. Other layers, such as RNN, LSTM and GRU, require an even larger number of weights for implementation.

AlexNet [1] contains approximately 60M weights, encoded with 32-bits Floating Point (FP), and thus requires approximately 240MB memory space for storing the network weights. VGG19 [3] storage requirements are even larger. It contains approximately 138M weights, and requires 552MB memory space. More advanced DNNs, such as Inception V3 [5] and Inception V4 [6], are composed of approximately 26M weights and require 104MB memory space. Additional memory is needed for internal network temporal arithmetic computations. Because of these extensive memory requirements, deploying DNNs on mobile devices presents a unique challenge for low-power and efficient implementation in embedded systems with limited memory space.

The associate editor coordinating the review of this manuscript and approving it for publication was Junxiu Liu.

One of the common approaches to meet this challenge is to remotely process the data and deploy the DNN using a powerful remote cloud server. For example, the Google Translate application uses a remote LSTM DNN [8] for real-time text translating. The translation task is carried out by sending the text from the local device to the cloud for processing. However, this approach increases execution time and requires an expensive continuous communication line, which is not always feasible.

This paper proposes an efficient entropy-based lossless compression and decompression algorithms and a real-time hardware decoder implementation method. This method provides for a local solution to enable DNN implementation on edge devices. The proposed algorithms are efficiently applied to DNN weights compression to enable decoding one weight in one processor clock cycle during the decompression phase. The efficiency of the proposed hardware implementation is evaluated in terms of its compression ratio compared to the well-known 7ZIP lossless compression algorithms. Experimental results show a high compression ratio comparable to the 7ZIP algorithms, while maintaining the one-cycle decoding constraint. Moreover, the proposed approach enables a very significant reduction, up to 70%, in the required system memory.

The rest of the paper is organized as follows. Section II presents related DNN compression work. Section III describes the proposed method and an efficient hardware decoder implementation. Section IV demonstrates experiments and results. Section V summarizes the paper.

II. BACKGROUND AND RELATED WORK

Many hardware accelerators [9]–[12] have been proposed to efficiently implement DNNs. However, most of these approaches do not apply weights compression. Typical FPGA solutions focus on computation optimization and therefore lead to efficient processing time [10], [11]. Achararit *et al.* [13] propose a DNN design framework using weight sharing and reinforcement learning-based methodology to accelerate the process of DNN generation. Han *et al.* [12] develop a DNN engine that takes advantage of sparse matrices and achieves high throughput. A weights compression approach based on the Huffman algorithm is also proposed by Han *et al.* [14]. However, this approach does not provide a hardware solution for implementing the Huffman compression algorithm.

Several approaches for DNN compression have been proposed recently for addressing the limited memory in mobile devices [14]–[17].

D. Blalock *et al.* provide a comprehensive overview of approaches to pruning [18]. Typical pruning approaches suggest reducing the number of weights by reducing the DNN size, thereby minimizing required arithmetic operations [16]. Pruning can also be carried out by erasing the neuron connections for the less significant weights, resulting in sparser weights matrices [14].

Hao and Li [15] suggest deleting redundant kernels in the convolutional layers with minimal accuracy loss. They show that pruning of 50% of the kernels in some convolution layers of the VGG network results in a 34% FLOP reduction. Molchanov *et al.* [16] suggest a new formulation for pruning convolutional kernels based on Taylor expansion that approximates the change in the cost function induced by pruning network parameters. They demonstrate high accuracy and a significant reduction of DNN complexity (approximately 50% of the required FLOPs in VGG).

A similar pruning approach is used in Deep Reinforcement Learning (DRL). Huang *et al.* [17], propose a DRL neural network, in which a reward is granted for each case in which a convolutional kernel is pruned without affecting the accuracy. They show that pruning 92.8% of the convolution kernels in VGG network results in only 3.4% accuracy loss. Ye *et al.* [19] propose a progressive weight pruning approach and demonstrate high pruning rate by using partial pruning with moderate pruning rates. Aghasi *et al.* [20] develop a convex post-processing technique that prunes a trained network layer by layer while preserving the internal responses.

These pruning approaches suggest lossy compression algorithms (generally referred to as different types of pruning). Because they change the uncompressed DNN structure, they may affect network accuracy and performance.

A deep compression method proposed by [14] suggests integrating both lossy (using pruning and quantization), and lossless compression (based on entropy). Results using this method show that AlexNet shrinks the memory required from about 240MB to only 6.9MB. Han *et al.* propose an entropy-based encoding of the weights using Huffman's algorithm [21]. However, the Huffman algorithm is complicated to implement in real-time and cannot meet the constraint of one-cycle decoding to avoid starvation. To ensure the decoding of one weight per clock cycle using the Huffman tree search, the clock frequency is limited by the time needed to reach the deepest leaf of the binary tree. This is not a practical solution for a real-time, one-cycle tree search, especially in a deep binary tree. For example, for 16-bit weights coding, a huge Huffman tree is generated with 64K nodes and at least 16 levels.

Pal *et al.* [22] propose a modified Huffman-based lossless compression technique for DNN weights compression. However, their method requires a top-down traversing of a complex deep Huffman binary tree.

Nevertheless, there are several approaches for hardware implementation of the Huffman algorithm [23], [24].

Mansour [23] proposes to implement the Huffman algorithm using a two-level Look Up Table (LUT). This approach uses multiple repetitions of shortcode words, and requires a large LUT (depending on the longest code-word). Therefore, it significantly increases the decoder memory space.

Dahri *et al.* [24] propose to implement Huffman algorithm on an FPGA. Since the search for a code-word match is carried out sequentially (reading bit by bit), this method can

not promise real-time, one-cycle decoding and may cause system starvation.

Some hardware implementations also exist for other lossless compression algorithms [25], [26]. Najmabadi *et al.* [25] propose a hardware architecture based on an asymmetric table decoding algorithm. The drawback of such an algorithm is that each symbol decoding may require more than one cycle. Moreover, the decoder hardware implementation requires about 144KB per a 16-bit code-word, while our proposed algorithm requires far less (only 16KB).

Lin and Chang [26] propose a two-stage lossless decompression algorithm that combines the PDLZW algorithm and an approximated adaptive Huffman algorithm. However, this algorithm cannot be implemented in real-time to ensure extracting a weight coefficient in one cycle.

To overcome the limitations of previous related works, we suggest a real-time oriented approach. The main contribution of the proposed algorithm is its ability to achieve a high compression rate, while enabling the extraction of one coefficient weight for every cycle. We propose an efficient hardware decoder implementation that can easily be integrated into a DNN hardware accelerator.

III. THE PROPOSED COMPRESSION APPROACH

The proposed compression algorithm is based on the Huffman algorithm and is applied to pre-partitioned weight classes according to the appearance probability of the weights. The Huffman algorithm is a Variable Length Coding (VLC) entropy-based compression algorithm. The code words are encoded into an appropriate binary tree, in which each leaf represents a code-word. The matching code-word process requires top-down traversing of the tree. Although this search can easily be implemented by software, it is complicated to implement on low-level hardware given memory limitations and strict real-time constraints. Moreover, Huffman trees usually are not balanced binary trees (representing multiple non-equal leaf levels). Therefore, the weights decoding time is not constant and depends on the leaf depth. The entropy of a given weights vector W is given by:

$$H(W) = -\sum_{w_i \in W} P(w_i) \cdot log_2 P(w_i) \tag{1}$$

where $P(w_i)$ is the probability of appearance of weight w_i. The entropy $H(W)$ reflects the theoretical minimal average number of bits required for encoding a weight in vector W.

Entropy-based VLC assigns different code length for each code-word, based on the symbol appearance probability. In our case, the symbols are represented by the DNN weights. Code-words are assigned to weights based on their appearance probability. Thus, different symbols can have different code lengths. A good entropy-based coding algorithm should converge to the entropy $H(W)$ in terms of the average number of bits required to encode a weight in vector W.

Fig. 1 shows the weights histogram in a typical DNN. The weights histogram is depicted for AlexNet (on CIFAR-10) and is represented by a normal distribution

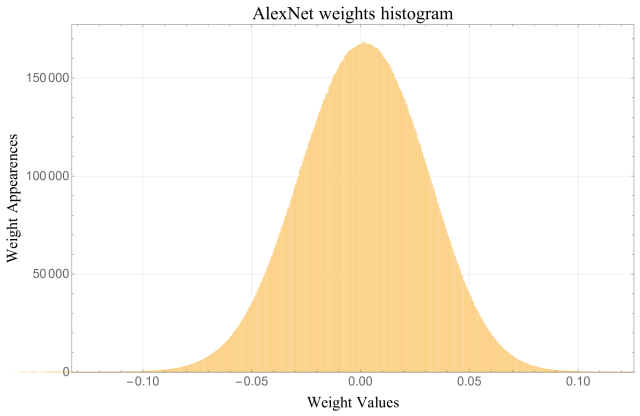


FIGURE 1. AlexNet weights histogram.

with a zero expectation. The appearance probability of each weight can be easily derived from the histogram plot. As part of the compression process, each weight in the DNN is assigned a new VLC code-word according to its appearance probability.

As mentioned before, hardware implementation of the Huffman algorithm, under strict real-time and memory constraints, is not trivial. Implementing an unbalanced Huffman tree using a single LUT yields high redundancy and is not practical due to the huge memory required.

To avoid a complex traversing on the Huffman tree, we suggest grouping the DNN weights into classes according to their appearance probability. Weights with a similar probability of appearance are assigned the same class, resulting in a small number of classes compared to the number of weights. The Huffman algorithm is then applied to the weight classes, representing a relatively simple tree that can be implemented by a small LUT.

To achieve efficient class coding, each class size is assigned a power of 2, except for a specific *Residual* class that contains the weights with the lowest appearance probability and can be any size.

To decode a specific weight, the class assigned to that weight is decoded using the Huffman algorithm implemented by a relatively small LUT. Then simple second level decoding is applied to extract the actual weight.

Applying the Huffman coding to a relatively small number of classes, instead of coding a large number of DNN weights, results in a simple and practical representation of the Huffman tree.

A. THE PROPOSED LOSSLESS COMPRESSION ALGORITHM

To meet the strict constraint of real-time weight decoding in one-cycle, the compression algorithm must be simplified, compromising on the quality and efficiency of the selected compression method. Selection of the proposed low-complexity compression algorithm results in a trade-off between real-time implementation and the compression ratio. Although other entropy-based weight encoding using

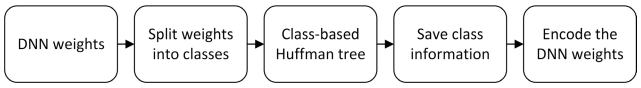


FIGURE 2. Compression algorithm flow.

FIGURE 3. Weight encoding structure.

Huffman's algorithm may yield better compression, it is too complicated to implement in real-time and cannot meet the constraint of one-cycle.

The compression algorithm's flow is described in Fig. 2. The algorithm contains four main stages: (a) splitting the weights into classes, (b) generating a class-based Huffman tree, (c) storing the tree structure, and (d) encoding of the weights. In the first phase, the DNN weights are divided into classes according to their appearance probability. To achieve an efficient entropy-based compression ratio, the number of bits that should be assigned to a specific weight w_i is derived from the entropy Eq. 1 as follows:

$$Bits(w_i) = Round(log_2(P(w_i))) \tag{2}$$

The weights are sorted in descending order according to their appearance probability. Then, the required number of bits is calculated for all sorted DNN weights using Eq. 2. The main idea is to classify weights, which should be encoded with the same number of bits into the same class.

A greedy process is used to define the size of the first class (including the highest probability weights). First, the number of weights N is found for encoding the number of bits assigned based on the highest probability. Since the size of the class is a power of 2 (due to hardware limitations), a rounding operation is needed. Therefore, the size of the first class is calculated as follows:

$$\textbf{Class Size} = 2^{Round(log_2 N)} \tag{3}$$

This process is repeated for all unclassified weights until all the weights are classified. The appearance probability for each class $C_j \in Classes$ is defined as the sum of all appearance probabilities of the weights in that class, and is calculated as follows:

$$P(C_j) = \sum_{w_i \in C_j} P(w_i) \tag{4}$$

The Huffman algorithm is applied to all classes, resulting in a specific code that is assigned to each class. The length of the code assigned to each class is proportional to the class probability. A class with high probability has a relatively short code.

The weights in each class are grouped in descending order according to their appearance probability. Each weight is assigned an *Index* according to its probability (i.e. *Index* zero is assigned to the weight with the highest probability in the class).

Fig. 3 shows the compressed weight encoding structure. The coding is composed by concatenation of two fields: *Class* and *Index* codes. The *Class* code is actually the generated Huffman code. The length of the *Index* code is determined by the class size, as follows:

$$\textbf{Index Length} = log_2(\text{Class Size}) \tag{5}$$

The compression process is completed after replacing the entire weights by their resulting compressed code and storing the compressed vector in SRAM external memory.

Information regarding the class, including Huffman *Class* codes and all class sizes, is stored in the internal decoder memory for further use in the decoding stage.

B. A DETAILED EXAMPLE OF THE COMPRESSION ALGORITHM

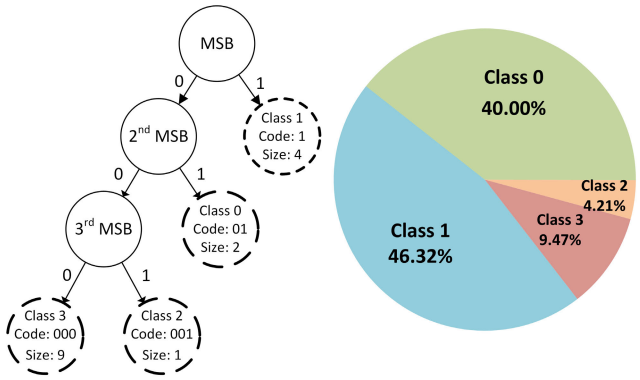
This section describes the proposed algorithm step by step. Let's assume that the weight vector represents a DNN network containing 95 weights with 4 -bits each. To simplify the example, we assume that there are only 16 different weights, and therefore only 4-bits are assigned to each weight. Table 1 demonstrates the probability of appearance, the calculated entropies, and the required number of bits for each of the unique 16 weights (out of the total 95 weights). The weight with the highest probability "0011", appears 20 times, and have the lowest entropy, leading to a 2-bit representation. The weights with the lowest appearance probability (appear only once) require 7-bit representation. The weights are sorted in descending order according to their appearance probability. The weights are split into four classes, encoded with a different number of bits per class.

TABLE 1. Split weights into classes example.

Weight	Appearances	$P(w_i)$	$-log_2 P(w_i)$	Bits
0011	20	21.05%	2.25	2
0110	18	18.95%	2.4	2
0010	15	15.79%	2.66	3
0111	12	12.63%	2.98	3
1111	11	11.58%	3.11	3
0000	6	6.32	3.98	4
1100	4	4.21	4.57	5
0001, 0100, 0101, 1000, 1001, 1010, 1011, 1101, 1110	1 (x9)	1.05% (9.47%)	6.57	7

Afterwards, a class-based Huffman algorithm is performed on the above classes, unlike the compression algorithm presented in [14] that applies Huffman directly on the DNN weights. The appearance probability of each class is defined as the sum of all its weights probabilities. Fig. 4 depicts the class-based Huffman tree.

Fig. 4a specifies the assigned code for each class for the example given in Table 1, while Fig. 4b shows the probabilities of each class according to Eq. 4.



(a) Huffman Tree (b) Class Probability

FIGURE 4. Class-based Huffman Tree: (a) Huffman Tree, and (b) Class Probability.

TABLE 2. Weights encoding.

Weight	# Class	Class Code	Index Code	Encoding
0011	0	01	0	010
0110			1	011
0010	1	1	00	100
0111			01	101
1111			10	110
0000			11	111
1100	2	001	-	001
0001	3	000	0000	000_0000
0100			0001	000_0001
0101			0010	000_0010
1000			0011	000_0011
1001			0100	000_0100
1010			0101	000_0101
1011			0110	000_0110
1101			0111	000_0111
1110			1000	000_1000

According to the Huffman tree, each of the 16 weights is assigned with a unique *Class* code and an *Index* code, as depicted in Fig. 4. Table 2 describes the weight encoding according to the proposed algorithm. While weights with high probability are encoded with only 3 bits, weights with low probability are encoded with 7-bits. The compression process results in a total of 321 bits required for weights representation, instead of 380 bits (95 x 4 bits per weight). It is important to note that the resulting average number of bits per weight (3.379) is close to the entropy value (3.152).

C. THE PROPOSED DECOMPRESSION ALGORITHM

The compressed file is stored in external SRAM memory. The decompression phase restores the encoded weights in a sequential manner, deploying weight-by-weight decoding. First, the weight *Class* is decoded, according to the class-based Huffman tree. Then, the weight is directly retrieved from the internal decoder memory using the *Index* code. This process is repeated for all the weights in the compressed weights vector. The next sections explain the structure of the internal decoder memory, hardware limitations, and the proposed efficient hardware implementation. The complexity of the proposed decompression algorithm is O(1), assuming a low number of classes (up to 16), compared to Huffman $O(log(n))$ complexity.

1) ENCODER/DECODER LOOK UP TABLES

The internal decoder memory is composed of three fixed-size LUTs: LUT1, LUT2, and LUT3. These tables are used for storing the Huffman tree, class information, and the decompressed weights, respectively. LUT1 contains the *Class* numbers addressed by the Huffman *Class* code, as shown in Table 3. The size of LUT1 is determined by the longest *Class* code and the number of classes. For example, in case the weight encoding results in 4 classes and a maximum 3-bit representation per *Class*, the LUT1 size is 2^3 x 2^2-bits.

TABLE 3. LUT1 Example (Huffman LUT).

Input 3-bit Addresses	Output 2-bit (Class Number)
000	11 (Class 3)
001	10 (Class 2)
010-011	00 (Class 0)
100-111	01 (Class 1)

TABLE 4. LUT2 Example (Classes LUT).

LUT1 output bits	LUT2 output		
	Class Code Length (minus 1)	Index Code Length	Class Offset Address
00 (Class 0)	01 (Length 2)	001 (Length 1)	000 (Address 0)
01 (Class 1)	00 (Length 1)	010 (Length 2)	010 (Address 2)
10 (Class 2)	10 (Length 3)	000 (Length 0)	110 (Address 6)
11 (Class 3)	10 (Length 3)	100 (Length 4)	111 (Address 7)

LUT2 stores three fields for each class: *Class* code length, *Index* code length, and class *Offset* (for addressing LUT3). LUT2 is addressed by the *Class* number derived from LUT1 output. Table 4 presents LUT2 organization for the detailed example given in the previous section (Sec 3-B).

TABLE 5. LUT3 Example (Weights LUT).

# Class	Address	Weight ($P(w_i)$)
0	0000	0011 (21.05%)
	0001	0110 (18.95%)
1	0010	0010 (15.79%)
	0011	0111 (12.63%)
	0100	1111 (11.58%)
	0101	0000 (6.32%)
2	0110	1100 (4.21%)
3	0111	0001 (1.05%)
	1000	0100 (1.05%)
	1001	0101 (1.05%)
	1010	1000 (1.05%)
	1011	1001 (1.05%)
	1100	1010 (1.05%)
	1101	1011 (1.05%)
	1110	1101 (1.05%)
	1111	1110 (1.05%)

LUT3, as shown in Table 5, stores the ordered uncompressed weights, addressed by adding the weight *Index* code to the class *Offset* (output of LUT2).

2) THE PROPOSED HARDWARE DECODER

The following parameters have been considered for evaluating the proposed decoder: (1) the size of LUT3, which stores the most common weights, (2) the number of classes used for classifying the weights according to their appearance probability, and (3) the maximal code length for encoding a class. In the experimental analysis, the size of LUT3 has been chosen to be 8 KB, with up to 16 classes and 8-bit maximal code length per class.

The size of LUT1 is determined by the longest Huffman *Class* code. For example, 16 classes and a maximum code length of 8-bits, requires LUT1 size of 2^8x2^4 bits. In this case, LUT2 has 16 entries and its actual size depends on the size of LUT3. To minimize the required internal memory, only part of the uncompressed weights, those with the highest appearance probability, are stored internally in LUT3. The remaining weights, characterized by low probability, are grouped into a *Residual* class, and derived directly from the compressed code using the *Index* code. Therefore, a different encoding structure is used.

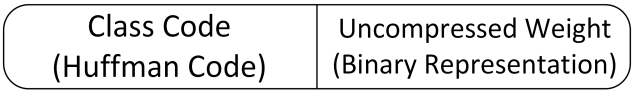


FIGURE 5. Residual class weights encoding.

Fig. 5 shows the modified coding structure of the *Residual* class weights. This coding is composed of two fields: (a) *Class* code, and (b) the uncompressed weight. As before, the *Class* code represents the Huffman code. Although each of these weights requires more bits than the original uncompressed 16-bit weight (16-bits uncompressed weight plus the *Residual* class code), it is not stored internally. Since the appearance probability of the *Residual* class is relatively low, the compression ratio is not reduced significantly, while still saving substantial memory space in LUT3. Table 6 shows an example for the assigned *Residual* weights encoding.

TABLE 6. Residual weights modified encoding.

Weight	# Class	Class Code	Modified Encoding
0001			000_0001
0100			000_0100
0101			000_0101
1000			000_1000
1001	3	000	000_1001
1010			000_1010
1011			000_1011
1101			000_1101
1110			000_1110

The number of the weight classes is determined empirically considering the available internal memory space. A higher compression ratio can be achieved by using a larger internal memory and more weights classes. However, the available internal memory of embedded devices is a major constraint. Thus, the number of weights classes is a tradeoff between internal memory size and the required compression ratio. Using 16 weights classes results in a high compression ratio (up to 70%) and relatively low LUT memory capacity (8KB).

For example, for 16-bit weights, an LUT3 size of 2^{16} words (i.e., 128 Kbytes) is required. However, results show that storing only 4096 weights internally has a minor effect on the compression ratio. The iterative process for splitting the weights into classes, as described in the compression phase, is completed by filling LUT3 with a predefined number of weights with high appearance probability. These weights should be stored internally.

Since only weights with high appearance probability are stored internally, a significant reduction in the LUT size is achieved (8KB compared to 128KB). LUT3 is implemented using 8KB SRAM, while LUT1 and LUT2 are implemented, due to their small size, using ARM register files that require a negligible area.

For example, implementation of a single 128KB SRAM using a 22nm process with ARM memory (type a_LL_SR_32768 × 32_M16B8_xaohwd) requires silicon area of $221,151um^2$, while implementing 8KB SRAM requires only $17,500um^2$ (using a a_LL_SR_4096 × 16_M8B2_xaohd of ARM). Therefore, a LUT area reduction of about 92% is achieved using the proposed method.

Algorithm 1 Split Weights Into Classes

Input: W - Sorted DNN weights
C_{max} - Maximum number of classes
LUT_{size} - Maximum number of weights to store in LUT3
Output: C - Classes of weights

$C_{number} \leftarrow 0$ ▷ class number initialization
$C_{offset} \leftarrow 0$ ▷ class offset initialization
while W is not empty **do**
  $C_{size} \leftarrow 1$
  $C_{bits} \leftarrow Bits(W[0])$ ▷ using Eq. 2
  while $C_{bits} = Bits(W[C_{size}])$ **and**
  $C_{size} + C_{offset} < LUT_{size}$ **do**
    $C_{size} \leftarrow C_{size} + 1$ ▷ increase the Class size
  end while
  $C_{size} \leftarrow 2^{Round(log_2(C_{size}))}$ ▷ round Class size
  if $C_{size} + C_{offset} > LUT_{size}$ **or** $C_{num} = C_{max} - 1$
    $C_{size} \leftarrow Len(W)$
  end if
  Define $W[0 : C_{size} - 1]$ **as a new class in** C
  Remove $W[0 : C_{size} - 1]$ **from** W
  $C_{offset} \leftarrow C_{offset} + C_{size}$ ▷ calculate next class offset
  $C_{number} \leftarrow C_{number} + 1$ ▷ calculate next class number
end while
return C

Algorithm 1 describes the process of splitting the weights into several classes. The following parameters serve as the

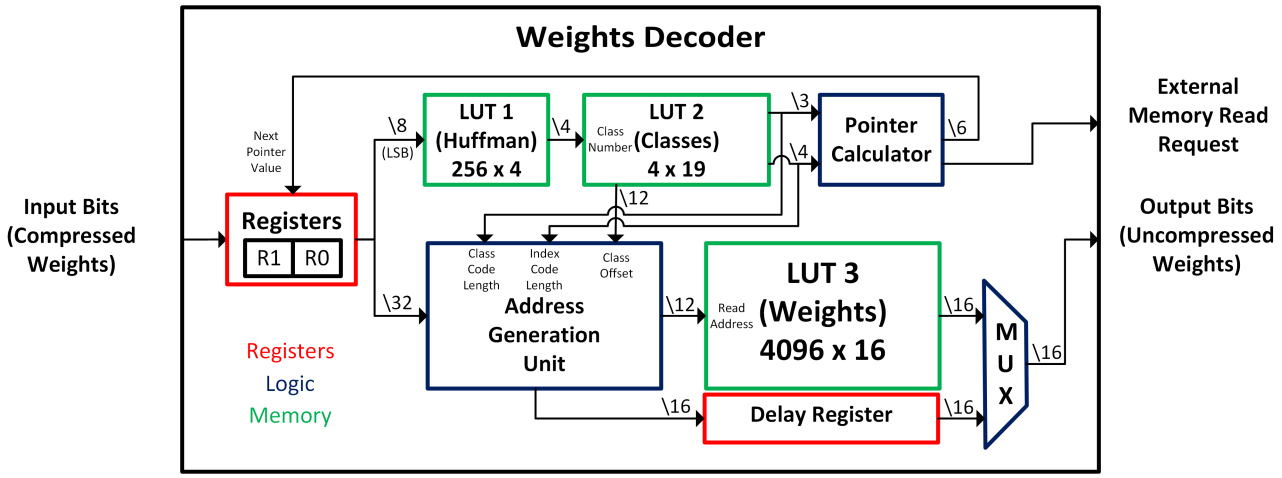


FIGURE 6. Hardware weights decoder block diagram.

algorithm input: the sorted DNN weights, maximum number of classes, and maximum number of weights to be stored in LUT3. The classification process starts with class initialization (lines 1-2). Lines 3-17 describe the first-class generation (as detailed in Sec. 3-A). This iterative process stops when one of the two following conditions exists: (a) the number of classes is greater than the predefined maximum (lines 11-13), or (b) LUT3 overflows (the number of class weights is greater than LUT3 capacity). Finally, the remaining weights are unified in the *Residual* class and the algorithm returns the resulting class partitioning.

D. HARDWARE DECODER IMPLEMENTATION

Real-time implementation requires usage of extra internal memory. To minimize this, only part of the uncompressed weights, those with the highest appearance probability, are stored internally. The remaining weights, characterized by low probability, are grouped into unique *Residual* class and are extracted directly from external memory. A dedicated ASIC (with 8KB internal SRAM) has been developed to implement the proposed decoder.

Fig. 6 depicts hardware decoder implementation, composed of four main modules: a register file containing two registers, three LUTs, address generation unit (AGU), pointer calculator, and some extra logic. The input to the decoder is a compressed 32-bit data, while its output represents a 16-bit decompressed weight. The decoder is designed to decompress one 16-bit weight for each cycle. Since we use VLC coding, the compressed weights are of different sizes. The code length of the compressed weight is derived from LUT2 and calculated by a dedicated pointer. Registers (R0 and R1) are used as double buffers, enabling sampling of new input while processing the previous one. The 32-bit input is always directed to R1 and copied to R0 initially every time the pointer crosses the value of 31 (in this case, the pointer generates a read request from the external SRAM).

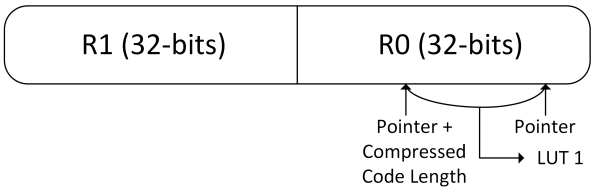


FIGURE 7. Registers bits concatenation into 64-bit buffer.

The 8-bit input to LUT1 is extracted from registers R0 and R1 (Fig. 7), and the specific compressed weight is addressed by the pointer. The appropriate *Class* number (LUT1 output) serves as input to LUT2. The three LUT2 data fields (length of the *Class* and *Index* codes, and class *Offset*) are used to extract the *Index* code from the compressed code and update the pointer to address the next compressed weight code for the next cycle. The AGU generates the entry address for LUT3 by adding the *Index* code to the class *Offset*. In case the *Class* number represents the *Residual* class, the AGU outputs the uncompressed 16-bit weight (retrieved from the *Index* field) directly to a one-cycle delay register. Additionally, an output control line is used for selecting the appropriate output from LUT3 or from the delay register, using a MUX.

Fig. 8 depicts the decoder timing diagram. The module is implemented using a 4-stage pipeline: (a) class decoding (b) address generation (c) weight extraction, and (d) weight output. The proposed hardware architecture enables real-time decoding of one weight at each clock cycle at 125 MHz.

The uncompressed (decoded) weights are either used only once in an FC layer, with no need for temporary storage, or they are reused in a Conv layer, requiring a small temporary buffer. This internal buffer is dedicated to storing the temporary partial uncompressed weights for a single filter only. Its actual size is negligible compared to the size of the overall CNN weights size. For example, for AlexNet with a $3 \times 3 \times 384$ convolution filter and 16-bit weights, the size of this additional temporary buffer is only about 7KB memory.

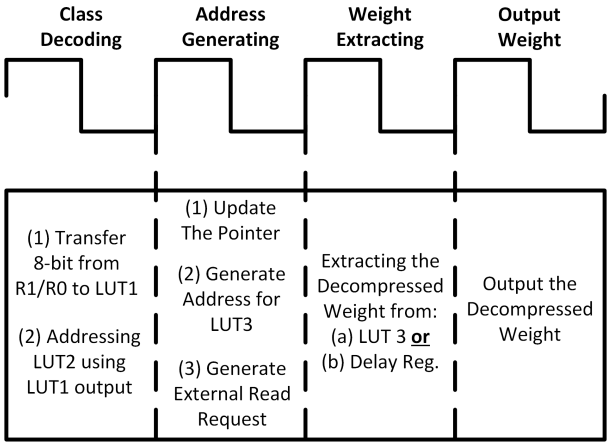


FIGURE 8. Four pipeline stages decoder.

This unique hardware implementation enables compressed weights decoding in one cycle within the original DNN timing. There is no extra CPU stalling and starvation is avoided. The additional decompression phase is integrated into the overall DNN architecture and is performed in parallel to the weight fetching stage using the pipeline approach. Therefore, the proposed approach can be applied to other more complex DNN architectures (such as: ResNet and MobileNet) while maintaining the real-time constraint. It is important to note that the end-to-end execution time is not in any way affected by applying the proposed compression approach, nor is the original DNN processing time (without compression) lengthened.

IV. EXPERIMENTAL AND RESULTS

The proposed approach has been evaluated using two well-known DNN architectures: AlexNet and VGG19. Each network has been tested with two benchmark datasets: MNIST, CIFAR-10 for AlexNet and CIFAR-10, CIFAR-100 for VGG19. AlexNet has been implemented using 5 Conv layers (with 3×3 kernels) and 3 FC layers, while VGG has 14 Conv layers (with 3×3 filters) and 3 FC layers. For both networks, a batch normalization has been applied, and the dropout mechanism has been used in the training process to eliminate overfitting. The training phase has been carried out using the Tensorflow framework and optimization tool [27]. Each of the DNNs has been trained, with the two datasets, using four different final sparsity configurations (60,70, 80 and 90%).

To evaluate the efficiency of the proposed lossless DNN weights compression algorithms, we refer to the DNN weights matrixes generated after completion of the training phase. Both AlexNet and VGG19 result in high top-1 accuracy for all datasets and are comparable to previously published results [1], [28], shown in Table 7.

The DNN weights of all the network layers are quantized (by asymmetric quantization) and concatenated into one long weights vector W in a predefined order. Then, W is compressed using the proposed algorithm, demonstrating a high compression ratio of up to 70%. For validating the correctness of the algorithm, a decompression phase is applied, providing exactly the same input weights vector W without data loss.

TABLE 7. DNN accuracy comparison Table.

DNN	Dataset	Reference Top-1 Accuracy	Our Top-1 Accuracy
AlexNet	MNIST	99.7	99.5
	CIFAR-10	87	78.5
VGG19	CIFAR-10	87	83.87
	CIFAR-100	58	52.68

The compression capabilities of our proposed approach were compared to several lossless compression algorithms presented by the 7ZIP package [29].

For the default 7ZIP compression algorithm, LZMA, a decompression throughput of about 10-20Mbps is typically obtained using a 2GHz CPU [29]. Therefore, the decompression of one weight in LZMA requires about 100 cycles. The proposed method demonstrates a relatively high compression ratio comparable to the 7ZIP algorithms while maintaining the one-cycle decoding major constraint.

For each experiment, the appropriate DNN weights vector was compressed using five popular algorithms from the 7ZIP package: Deflate, LZMA, LZMA2, PPMd and BZip2. The algorithm that yielded the best compression, has been chosen as a reference for comparison with our algorithm. The compression ratio is defined by Eq. 6.

$$\textbf{Compression Ratio} = 1 - \frac{Compressed}{Uncompressed} \tag{6}$$

Since the decoder internal LUT sizes are negligible compared to the size of the compressed weights, the compression ratio has been evaluated considering only the size of the compressed file, as described in Eq. 6.

TABLE 8. AlexNet compression results on MNIST and CIFAR-10 with various sparsity levels.

Dataset (Weights File Size)	Sparsity	Top-1 Accuracy	Best 7Zip Algorithm Compression Ratio	Our Algorithm Compression Ratio	Our Algorithm Compression Result
MNIST (40.45 MB)	0.00%	99.40%	-	-	-
	37.79%	99.32%	31.48%	30.63%	28.06 MB
	42.17%	99.41%	35.66%	34.71%	26.41 MB
	49.26%	99.49%	42.46%	41.50%	23.66 MB
	56.86%	99.40%	50.25%	49.52%	20.42 MB
CIFAR-10 (46.46 MB)	0.00%	78.26%	-	-	-
	52.47%	78.50%	45.94%	44.98%	25.26 MB
	58.46%	78.20%	50.67%	51.24%	22.65 MB
	66.84%	78.23%	60.22%	60.26%	18.46 MB
	75.63%	78.37%	69.57%	68.72%	14.53 MB

Table 8 and Table 9 show the results for AlexNet and VGG19 for the different databases using various sparsity levels. Results demonstrate a high compression ratio, and are comparable to the best 7ZIP algorithm, while our algorithm uniquely allows decompression of one weight per clock cycle. For example, for classifying CIFAR-10 using AlexNet with 75.63% sparsity, the best 7ZIP algorithm results in 69.57%, while our proposed algorithm achieves close results of 68.72%.

TABLE 9. VGG19 compression results on CIFAR-10 and CIFAR-100 with various sparsity levels.

Dataset (Weights File Size)	Sparsity	Top-1 Accuracy	Best 7Zip Algorithm Compression Ratio	Our Algorithm Compression Ratio	Our Algorithm Compression Result
CIFAR-10 (72.06 MB)	0.00%	83.87%	-	-	-
	44.00%	82.40%	37.66%	36.11%	46.04 MB
	58.47%	83.23%	51.45%	51.03%	32.59 MB
	68.61%	83.31%	61.90%	61.31%	27.88 MB
	70.24%	83.15%	63.56%	62.99%	26.67 MB
CIFAR-100 (72.77 MB)	0.00%	52.68%	-	-	-
	40.07%	52.60%	34.62%	32.74%	48.94 MB
	47.30%	51.58%	40.71%	39.13%	44.29 MB
	53.07%	50.88%	46.36%	45.15%	39.91 MB
	62.79%	51.60%	55.98%	55.20%	32.60 MB

Similar results are achieved for classifying CIFAR-10 using VGG19 with 70.24% sparsity, demonstrating a 63.56% compression ratio for the best 7ZIP algorithm, and 62.99% using the proposed algorithm.

V. CONCLUSION

This work presents an entropy-based compression algorithm and a real-time oriented decompression approach. The proposed unique and efficient hardware decoder implementation, allows decompression of one weight in a single clock cycle. Results show a high compression ratio compared to well-known lossless compression algorithms. The achieved compression ratio is comparable to the 7ZIP compression package and enables efficient implementation of large DNNs in a low-power SoC with limited memory resources. The combination of the existing TensorFlow built-in pruning tool with our proposed compression algorithm may yield better results as sparsity increases. The proposed approach was evaluated using different DNNs and various image datasets, demonstrating consistent results and promising robustness. Future research can integrate compression of the inter-layer data, in addition to the weights compression, to further reduce the required internal memory for implementing DNN in edge devices.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. New York, NY, USA:Curran Associates, 2012, pp. 1097–1105.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: https://arxiv.org/abs/1409.1556

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[6] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. ICLR Workshop*, 2016, pp. 1–7.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2014.

[8] Y. W. *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*. [Online]. Available: https://arxiv.org/abs/1609.08144

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[10] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays FPGA*, 2015, pp. 161–170.

[11] C. Liu, C. Wang, and J. Luo, "Large-scale deep learning framework on FPGA for fingerprint-based indoor localization," *IEEE Access*, vol. 8, pp. 65609–65617, 2020.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network.," in *Proc. ISCA*, Jun. 2016, pp. 243–254.

[13] P. Achararit, M. A. Hanif, R. V. W. Putra, M. Shafique, and Y. Hara-Azumi, "APNAS: Accuracy-and-performance-aware neural architecture search for neural hardware accelerators," *IEEE Access*, vol. 8, pp. 165319–165334, 2020.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149

[15] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*. [Online]. Available: https://arxiv.org/abs/1608.08710

[16] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *Proc. 5th Int. Conf. Learn. Represent. ICLR*, Toulon, France, Apr. 2017, pp. 1–17.

[17] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," in *Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV)*, Mar. 2018, pp. 709–718.

[18] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning," in *Proc. Mach. Learn. Syst.*, 2020, pp. 129–146.

[19] S. Ye, T. Zhang, K. Zhang, J. Li, K. Xu, Y. Yang, F. Yu, J. Tang, M. Fardad, S. Liu, X. Chen, X. Lin, and Y. Wang, "Progressive weight pruning of deep neural networks using ADMM," 2018, *arXiv:1810.07378*. [Online]. Available: https://arxiv.org/abs/1810.07378

[20] A. Aghasi, A. Abdi, and J. Romberg, "Fast convex pruning of deep neural networks," *SIAM J. Math. Data Sci.*, vol. 2, no. 1, pp. 158–188, Jan. 2020.

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

[22] C. Pal, S. Pankaj, W. Akram, A. Acharyya, and D. Biswas, "Modified Huffman based compression methodology for deep neural network implementation on resource constrained mobile platforms," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[23] M. F. Mansour, "Efficient Huffman decoding with table lookup," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. - ICASSP*, Apr. 2007, pp. II–53–II–56.

[24] S. A. Dahri, A. F. Chandio, and N. A. Zardari, "Implementation of Huffman decoder on fpga," *Int. J. Eng. Res. Appl.*, vol. 6, no. 1, pp. 84–88, Jan. 2016. [Online]. Available: www.ijera.com

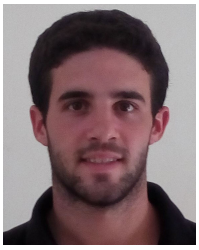
[25] S. M. Najmabadi, H. S. Tungal, T.-H. Tran, and S. Simon, "Hardware-based architecture for asymmetric numeral systems entropy decoder," in *Proc. Conf. Design Archit. Signal Image Process. (DASIP)*, Sep. 2017, pp. 1–6.

[26] M.-B. Lin and Y.-Y. Chang, "A new architecture of a two-stage lossless data compression and decompression algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 9, pp. 1297–1303, Sep. 2009.

[27] TensorFlow. *Pruning With Keras*. Accessed: Oct. 2020. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras

[28] F. He, T. Liu, and D. Tao, "Control batch size and learning rate to generalize well: Theoretical and empirical evidence," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds. New York, NY, USA: Curran Associates, 2019, pp. 1143–1152.

[29] D. Salomon and G. Motta, *Handbook of Data Compression*, 5th ed. London, U.K.: Springer-Verlag, 2010.



TOMER MALACH received the B.Sc. degree (Hons.) in electrical and computer engineering from the Ben-Gurion University of the Negev, Be'er Sheva, Israel, in 2019. His research interests include hardware architecture, VLSI design, machine learning, and data compression.

SHLOMO GREENBERG (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. degrees in electrical and computer engineering from the Ben-Gurion University of the Negev, Be'er Sheva, Israel, in 1976, 1984, and 1997, respectively. He is currently a Staff Member with the School of Electrical and Computer Engineering, Ben-Gurion University of the Negev. His main research interests include computer architecture, machine learning, image and digital signal processing, computer vision, and VLSI low-power design.

MOSHE HAIUT received the B.Sc. degree (Hons.) in electrical engineering from Technion, Israel, and the M.Sc. degree (Hons.) in electrical engineering from Tel-Aviv University, Israel. He is currently a Senior Engineer with DSP Group Ltd., Herzliya, Israel. His main research interests include communication, video, and audio digital signal processing, digital hardware architecture, and neural networks.