Combining Weight Approximation, Sharing and Retraining for Neural Network Model Compression

Prachi Kashikar, Olivier Sentieys, Sharad Sinha

Combining Weight Approximation, Sharing and Retraining for Neural Network Model Compression

PRACHI KASHIKAR, Indian Institute of Technology Goa, Ponda, India
OLIVIER SENTIEYS, INRIA, University of Rennes, Rennes, France
SHARAD SINHA, Computer Science, IIT Goa, Ponda, India

Neural network model compression is very important to achieve model deployment based on the memory and storage available in different computing systems. Generally, the continuous drive for higher accuracy in these models increases their size and complexity, making it challenging to deploy them on resource-constrained computing environments. This article proposes various algorithms for model compression by exploiting weight characteristics and conducts an in-depth study of their performance. The algorithms involve manipulating exponents and mantissa in the floating-point representations of weights. In addition, we also present a retraining method that uses the proposed algorithms to further reduce the size of pre-trained models. The results presented in this article are mainly on BFloat16 floating-point format. The proposed weight manipulation algorithms save at least 20% of memory on state-of-the-art image classification models with very minor accuracy loss. This loss is bridged using the retraining method that saves at least 30% of memory, with potential memory savings of up to 43%. We compare the performance of the proposed methods against the state-of-the-art model compression techniques in terms of accuracy, memory savings, inference time, and energy.

CCS Concepts: • **Hardware** → *Hardware-software codesign*; **Hardware accelerators**; • **Computer systems organization** → **Neural networks;**

Additional Key Words and Phrases: Model compression, exponent sharing, mantissa approximation, exponent share aware retraining

ACM Reference Format:
Prachi Kashikar, Olivier Sentieys, and Sharad Sinha. 2024. Combining Weight Approximation, Sharing and Retraining for Neural Network Model Compression. *ACM Trans. Embedd. Comput. Syst.* 23, 6, Article 99 (September 2024), 23 pages. https://doi.org/10.1145/3687466

1 Introduction

The demand for **artificial intelligence (AI)** has pushed toward bigger and complex neural network models with millions or billions of parameters. Such a large number of parameters has required megabytes of memory for on-chip or off-chip storage. The usage of more and more compute and memory resources for neural networks is not aligned with resource-accuracy tradeoff, which is one of the fundamental concepts in computing systems design. Hence, several research works

This work was supported by DST-INRIA-CNRS Project IFC/4131/DST-Inria/2018-2019/1 through CEFIPRA, India.
Authors' Contact Information: Prachi Kashikar, Indian Institute of Technology Goa, Ponda, India; e-mail: prachi183311004@iitgoa.ac.in; Olivier Sentieys, INRIA, University of Rennes, Rennes, France; e-mail: olivier.sentieys@inria.fr; Sharad Sinha, Computer Science, IIT Goa, Ponda, Goa, India; e-mail: sharad@iitgoa.ac.in.

ACM 1539-9087/2024/09-ART99
https://doi.org/10.1145/3687466

have focused on reducing the model size (i.e., model compression). Additionally, such memory or storage requirements can be a major limiting factor on mobile platforms, Internet of Things (IoT) devices, and so forth, which are generally computationally much less powerful, and where there is an ever-increasing interest in deploying AI models. Much commercial deployment on such devices is restricted to keyword spotting [39], audio-visual wake word detection [42], simple classification tasks, and so on because of the complexity and size of the larger models. At the same time, large compute environments consume high energy while processing neural network models. Hence, there is a critical need to develop model compression methods which would help address the challenges of deployment on smaller compute environments and energy dissipation.

The size and the complexity of a model depends on its number of layers, channels, weights, and activation functions. The higher performance demand increases the number of layers and complexity of the model architecture. Model training is typically done on high-end computers with powerful CPUs/GPUs, associated large on-chip and off-chip memories, and continuous power supply. Depending on the complexity of the model and performance requirements of the end application, training can take weeks to complete. Hence, existing model compression methods aim to reduce the model size with minimal error by modifying these elements. Model size can be reduced during training or post-training (i.e., by processing a trained model). Many methods focus on post-training compression because the generated inference models is still big, as we will see in Section 2.

It is challenging to compress a model without having any impact on accuracy. There are already known model compression techniques, as discussed in Sections 2.1, 2.2, and 2.3, that can be used to reduce the size of the models. These techniques can also be combined to increase memory savings. For example, in the work of Hong et al. [16], the size of the model is reduced to less than half with a combination of quantization and pruning.

The main contributions of this article are as follows:

— Algorithms that reduce the size of models by exploiting the characteristics of exponent and mantissa values in floating-point weights, and a systematic study and peer-comparison of these algorithms with respect to performance and accuracy tradeoff.
— A new **Exponent Share Aware Retraining (ESART)** algorithm that retrains pre-trained models, resulting in further reduction in model size.
— Comparison with prior work demonstrating that the proposed weight manipulation algorithms save at least 20% of memory on state-of-the-art image classification models with very minor accuracy loss. This loss is bridged using the retraining method that saves at least 30% of memory, with potential memory savings of up to 43%.

We believe that the systematic study of the proposed algorithms is important to identify the relative performance of these algorithms, which then creates a well-defined path forward for implementation.

The article is organized as follows. Section 2 provides a review of the existing techniques of model compression methods. This section discusses the prominent methods of pruning (Section 2.1), quantization (Section 2.2), and weight sharing (Section 2.3). In Section 3, the proposed algorithms for weight approximation are discussed. To address the accuracy loss due to the application of these algorithms, the proposed novel ESART approach is presented thereafter. Section 4 presents the results of the experiments conducted on different models using the proposed algorithms. A peer comparison among the proposed algorithms and with state-of-the-art methods is also presented. Section 5 concludes the proposed work with a discussion on directions for future work.

2 Related Work on Model Size Reduction

The precision of weights in a trained model plays a significant role in its accuracy. To achieve higher accuracy, the model needs to use higher precision, which, in turn, requires more storage space. Many techniques have been developed to reduce the precision of weights while minimizing the tradeoff with model performance. One way to reduce the model size is by reducing the number of weights, neurons, and channels, or a combination of these. Various popular techniques like weight sharing, quantization, weight pruning, knowledge distillation, and tensor decomposition are used to achieve this goal. These techniques can be applied during or after training to reduce the size of inference. The mainstream techniques that reduce model size are discussed in the following subsections.

2.1 Pruning

Weight pruning is a useful technique for reducing the size of a model, but it can also lead to some loss in accuracy. To mitigate this, fine-tuning the model after pruning is often necessary. One potential benefit of weight pruning is that it can speed up the training process by reducing the number of weights [13].

Different approaches propose various criteria to prune weights, such as energy-aware pruning [43], quantization-aware pruning [11], or performance-aware pruning [33]. It is possible to use the same pruning criteria for all layers in a model, but this general model-level pruning can lead to significant accuracy loss. To address this issue, Carreira-Perpinán and Idelbayev [4] demonstrate how to perform layer-wise pruning of weights. Weight pruning can be considered an optimization problem, where the weight that has the least impact on accuracy is pruned, resulting in minimal error after pruning.

In some approaches, filters that extract repetitive or less significant features with little impact on the accuracy of the model are removed. However, removing a whole filter may result in greater loss, which is why Meng et al. [28] propose a filter skeleton that divides filters into independent strips and forms new filters with optimal shapes. While filter pruning is more hardware-friendly, weight pruning results in a higher compression ratio.

Channel pruning is also used to reduce the model size. Other than selecting a fixed number of channels per layer, selective channel pruning has been demonstrated in the work of Guo et al. [12]. This reduces the drop in accuracy. This approach first prunes the network with other existing compression techniques. It then assesses the impact on accuracy and, in a reverse engineering manner, finds channels that have the least impact on accuracy. During the training phase, to reduce channels, the significance of every channel is observed and the one with the least importance is pruned in each epoch. This type of discrimination-aware channel pruning [25] during training saves a lot of memory bandwidth during inference.

2.2 Quantization

Neural networks often use floating-point weights, which are complex and expensive to compute with. To save memory and reduce complexity, weights can be quantized to low-bit fixed-point weights, resulting in a platform-independent inference [16]. The existing research work could quantize the weights to a single bit in a binary neural network [34], and such network can even facilitate training on the edge due to reduced memory footprint [40].

During quantization, weights are divided into fixed-sized bins, leading to a loss in accuracy during the reconstruction of original weights. This is because the fixed-size binning does not consider the significance of different weights. An adaptive integer quantization [41] method suggests selecting the best bin boundary instead of using a fixed-interval binning approach. This method

has shown relatively less loss in accuracy while achieving a fourfold model compression. Convolutional layers have higher redundancy in weights, and therefore they are less sensitive to quantization than fully connected layers. The adaptive quantization approach [6] for different layers that have dynamic shift and scale factors to reduce the accuracy loss can quantize state-of-the-art models like ResNet18/34/50, MobileNetV2, and EfficientNet-B0 to 4 bits.

The weight matrix has a different distribution of weights in every row. For example, in a sparse matrix, most of the weights are zero. In such cases, the quantization with fine granularity works best as it applies different quantization schemes for different rows of the same weight matrix [5]. One similar technique proposed in the work of Mishchenko et al. [29] does quantization in the column-wise manner, where every column in the weight matrix has its min-max range for quantization. Additionally, the outputs of every input frame are quantized individually rather than generalizing with specific hyper-parameters.

Post-training quantization results in models using parameters in fixed-point precision, whereas during training, floating-point precision is used. This conversion incurs an accuracy drop which can be prevented by training in fixed-point precision [27]. During training, such quantization reduces a lot of memory requirements for the weights. However, more memory is needed for activation functions. Quantization of activation also reduces the memory footprint during training significantly [24].

2.3 Weight Sharing

During training, there can be a lot of weight values close to each other. Such weight values can be replaced by a single value. This is often referred to as weight sharing during training. Model training time gets reduced with this type of weight sharing. Unlike pruning where weights are pruned, weight sharing shares them. Hence post-training, there is relatively less accuracy loss in weight sharing. Dupuis et al. [9] show how weight sharing can lead to more than four times the compression rate on state-of-the-art models like ResNet18 and SqueezeNet.

Weight sharing also regularizes the weights, and hence there is a low chance of overfitting during training. Like quantization, the weights are divided into different groups and the mean value and the spread of values are determined [38]. Depending upon the accuracy impact, these groups are altered. A different work by Dupuis et al. [8] proposes an automatic framework that explores the design space for sharing weights and finds the best approximate version for a given model.

3 Proposed Methods for Model Size Reduction

The size of the neural network model increases with an increase in the number of weights. These are generally floating-point weights. These weights are stored in memory using IEEE floating-point formats for different precisions as shown in Figure 1. Each floating-point weight is represented as a combination of sign, exponent, and mantissa.

A floating format with l bits of exponent can have 2^l values for the exponent. Models generally have millions of floating-point weights [31], whereas l is very small. Hence, there are many repetitive exponents. For example, Figure 2 shows the exponent frequency distribution in the weights of GoogleNet trained on the CIFAR10 dataset. In this case, weights are stored in the IEEE single-precision (Float32) floating-point format where the exponent is 8 bits wide. Out of 256 possible values, we can see from Figure 2 that less than 20 distinct exponents dominate the frequency distribution across different layers.

It is a good idea to share exponents in a layer-wise fashion as proposed in previous work [22]. This method, called *exponent sharing*, can save up to 10% of memory when weights are stored in Float32 precision, without affecting the accuracy of the model.

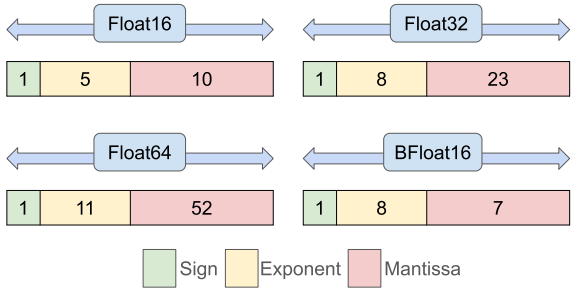


Fig. 1. IEEE floating-point number representations for different precisions. BFloat16 floating-point format is proposed by Google. It differs from IEEE Float32 only in terms of mantissa.

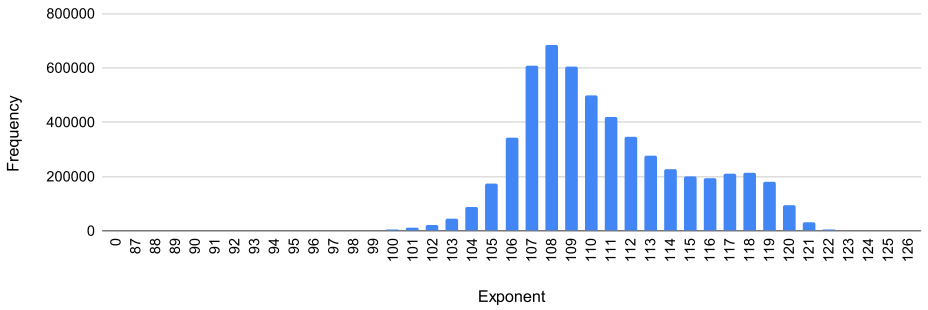


Fig. 2. Exponent frequency distribution in the weights of GoogleNet [37] trained on CIFAR10.

When using PyTorch for running models on processors, we rely on *torch.ldexp()* and *pow()*, which are used to transcode the compressed components of weights back to the standard floating-point formats. However, when using custom target hardware like **field-programmable gate arrays (FPGAs)**, we make use of the hardware architecture as proposed in our prior work [21], where separate tables are maintained for sign, index, and mantissa. Using an indexing mechanism, the values are read out from these tables and combined to form the floating-point weights which are then used by the DSP blocks in FPGA for performing the multiplication. In custom hardware like FPGA, it is possible to pipeline the combining of components into weight values and the multiplication operations such that the execution overhead during inference is very low, as can be seen in Section 4.5.

The compression ratio can be increased by reducing the number of distinct exponents and increasing their frequency. In this work, we propose several methods on top of the existing lossless exponent sharing technique, to approximate weights based on their exponent values and to achieve greater memory savings. We also discuss how mantissa approximation can boost memory savings in combination with exponent sharing. Figure 3 shows the overview of all methods discussed in this article. We discuss (1) exponent-based weight approximation by utilizing exponent magnitudes and exponent frequencies separately, (2) mantissa-based weight approximation, (3) exponent sharing by using (1) and (2), and (4) ESART that exploits mantissa-based weight approximation in (2).

3.1 Weight Approximation in Floating-Point Weights Using Exponents

Floating-point weights can be represented as a combination of sign, index, and mantissa, where index refers to the exponent stored separately in an exponent table [22]. When there are e distinct

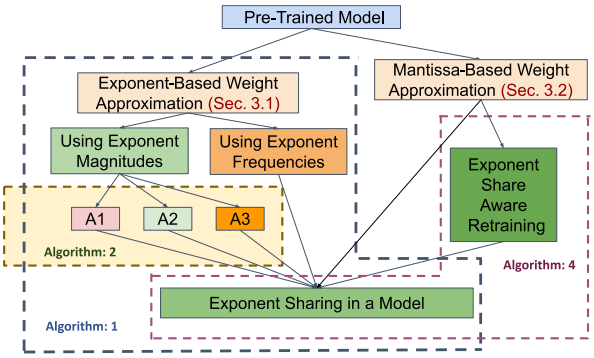


Fig. 3. Overview of all model size reduction methods studied in this article.

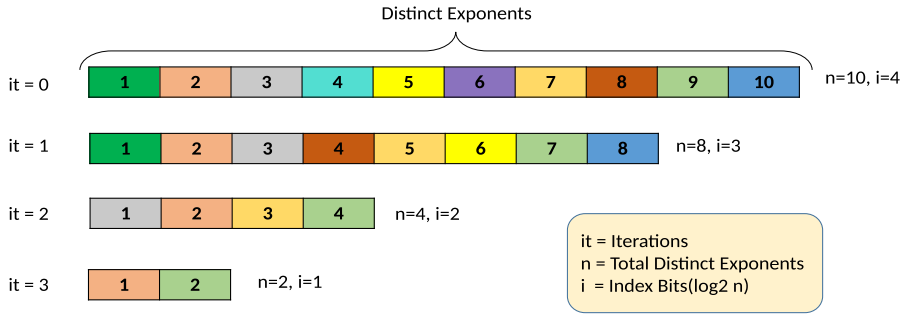


Fig. 4. Iterative exponent approximation in the floating-point weights.

values in the exponent table, $i = \lceil log_2(e) \rceil$ index bits are required. More memory is saved when e is smaller.

Our proposed method involves an iterative approach to weight approximation. In each iteration, we reduce the length of exponent table, as illustrated in Figure 4. This approach involves removing a few exponents from the table and approximating weights corresponding to those exponents. We focus primarily on approximating the smaller weights, as they have less impact on model performance [44]. To achieve this, we initially sort the exponent table of a layer in descending order. We then truncate this table in each iteration such that $i = i - 1$, where i is the number of index bits. The exponents that remain in the table are referred to as *salient exponents*, and the corresponding weights are referred to as *salient weights*. Our method focuses only on non-salient weights that have non-salient exponents. We approximate these weights using different methods shown in Algorithm 2, namely A1, A2, and A3. The illustration of all three methods is shown in Figure 5. Algorithm 1 shows the proposed algorithm of weight approximation. The iterations in Algorithm 1 consider one of the exponent-based weight approximation substitutes (*A1/A2/A3*) at a time as shown on line 12. This algorithm approximates the weights per layer in such a way that 1 bit less is required for indexing the exponent table after every iteration (lines 3–22). The iterations continue until either the compression reaches its maximum (or user-defined threshold) or the accuracy degradation exceeds a related user-defined threshold (line 18).

A1. Pruning Non-Salient Weights

From the sorted exponent table (e) in a layer, top $2^{\lceil log_2(e) \rceil - 1}$ salient exponents are selected. As like in magnitude-based pruning [10], all non-salient weights corresponding to non-salient exponents are made zero. This increases the sparsity in the weights of a layer.

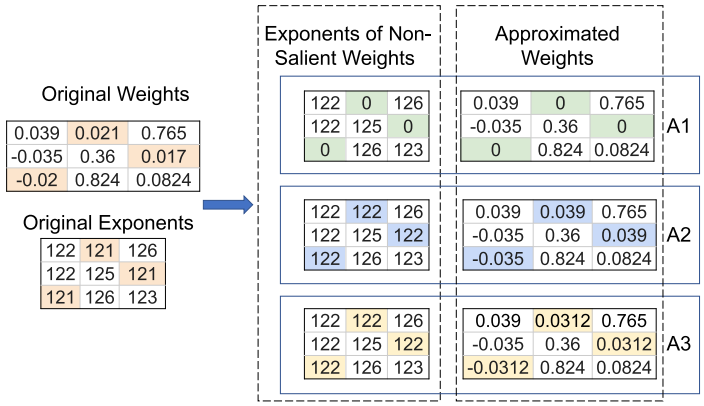


Fig. 5. Illustration of weight approximation using methods A1, A2, and A3 shown in Algorithm 2. The highlighted weights on the left are approximated by the three different methods on the right.

ALGORITHM 1: High-Level Description of Iterative Weight Approximation and Exponent Sharing

Input: Pre-Trained Model
Result: Compressed Model
Set the number of bits to reduce from the index field as $k = 1$;
Set memory savings (m_s) and accuracy drop (a_d) thresholds ;
while *layers* > 0 **do**
  Find distinct exponents (e) and generate an exponent table ;
  Find number of index bits $i = \lceil log_2(e) \rceil$ to refer the exponent table ;
  if $i < 3$ **then**
    continue;
  else
    Sort the exponent table;
    Choose top 2^{i-k} salient exponents `/* Weights having non-salient exponents are non-salient Weights */`
    while *non-salient weights* > 0 **do**
      Do weight approximation using method A1 or A2 or A3 from Algorithm 2;
    end
  end
end
Test the model and report the drop in accuracy (a) ;
Estimate memory saved (m) by model after exponent sharing using Algorithm 3 ;
if $m <= m_s$ *or* $a > a_d$ **then**
  Stop;
else
  $k = k + 1$;
  Go to step 3 for the next iteration ;
end

A2. Weight Adjustment without Changing the Salient Exponents

Pruning the weights significantly degrades the accuracy with the increasing number of iterations (related results are presented in Section 4.1.1). Instead of completely discarding the non-salient

ALGORITHM 2: Description of Weight Approximation Methods

```
Input: Weights, Non-salient weights
Result: Approximated Weight Tensor
n = number of non-salient weights ;
Salient weights = Weights − Non-salient weights;
Select weight approximation method (A1/A2/A3);
if A1 then
    while n > 0 do
        non-salient weight = 0 ;
    end
end
if A2 then
    while n > 0 do
        Find the root mean squared distance of the non-salient weight from other salient weights in a
          tensor ;
        p = Nearest salient weight for the given non-salient weight;
        non-salient weight = p ;
    end
end
if A3 then
    while n > 0 do
        Find the exponent of the non-salient weight ;
        Replace the non-salient exponent with nearest salient exponent in a tensor ;
        Make the mantissa zero ;
    end
end
```

weights, they are altered to become the nearest salient weights in the same layer. To find the nearest salient weights for every non-salient weight, the root mean square of the difference is calculated with all salient weights in that layer. As non-salient weights have lower exponents than salient weights, this approach results in every non-salient weight acquiring a value higher in magnitude.

A3. Weight Replacement Using Exponent-Mantissa Adjustment

In this method, we find the nearest salient exponent for the exponent of non-salient weights. As the exponent table is already sorted, the new exponent will be higher in magnitude. For any exponent, the smallest value that a floating-point weight can have is when the mantissa is zero. Therefore, we set the corresponding mantissa to zero to decrease the distance between the newly generated weight and the original non-salient weight.

In the context of this work, we consider the accuracy degradation maximum up to 10% to study the possible memory savings. After the last successful iteration of Algorithm 1, the weights are stored as shown in Figure 6. Hence, if the j iterations (lines 3–23 of Algorithm 1) are successful, then j bits are reduced from the index field. The memory saved after exponent sharing M_{comp} is given as

$$M_{comp} = N \times (s + i + m) + l \times e, \tag{1}$$

where N is the total weights in a layer, s is a sign bit, i are index bits, m are mantissa bits, e is the number of distinct values in an exponent table, and l is the length of the exponent field in the

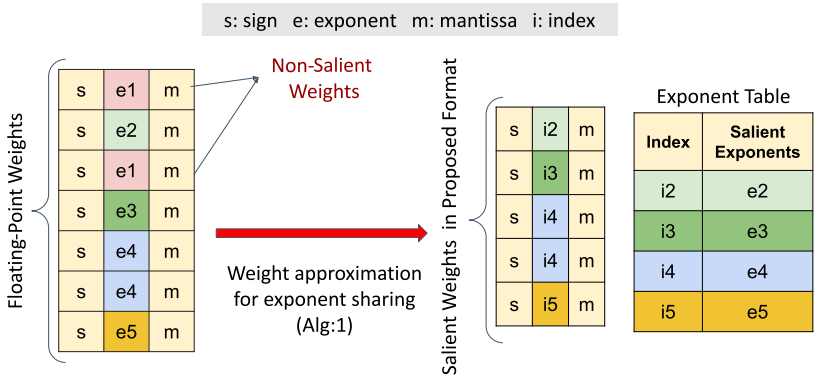


Fig. 6. Flow of proposed weight approximations and exponents sharing. Iterations of Algorithm 1 reduce the number of salient exponents as shown in Figure 4. The distinct exponents are shown with different color highlights.

ALGORITHM 3: Exponent Sharing in the Weights of Trained Models [21]

Data: Weight Tensor (W_i)
Result: Tensors of Sign (S_o), Index (I_o), Mantissa (M_o), and Exponent List (E_o)
Calculate memory requirement (M_{orig}) for W_i ;
while w *in* W_i **do**
  $S_o \leftarrow$ sign of w ;
  $E \leftarrow$ exponent of w ;
  $M_o \leftarrow$ mantissa of w ;
end
Select only distinct exponents from E and make an exponent table (E_o) ;
while e *in* E_o **do**
  Find Index i for e ; `// i is an index of e in` E_o
  Place Index i in I_o corresponding to e in E ;
end
Calculate memory requirement after exponent sharing (M_{comp}) using Equation (1) ;
Report memory savings (M_{orig} - M_{comp}) ;
Return S_o, I_o, M_o, and E_o

original floating-point format of the weight. The l will change with the precision of floating-point storage format (i.e., $l = 8$ for Float32 or $l = 5$ for Float16) as shown in Figure 1.

If M_{orig} is the memory required when the weights are stored in a standard floating-point format and M_{comp} is the memory required after exponent sharing, then the percentage of memory saved is expressed as

$$M_{savings} = 100 \times (M_{orig} - M_{comp})/M_{orig}. \tag{2}$$

Let us consider that the weights are stored in BFloat16 format. It has 8 bits to store the exponents. The maximum memory saved by exponent sharing happens when there are less than three distinct exponents ($e < 3$, $i = 1$), among all of the weights, because the exponent table is smallest in this case. Then the maximum percentage of memory saved ($M_{maxSavings}$) is

$$M_{maxSavings} = 100 \times ((16 \times N) - (N \times (1 + 1 + 7) + 8 \times 2)/16 \times N, \tag{3}$$

$$M_{maxSavings} = (43.75 - 100/N)\%. \tag{4}$$

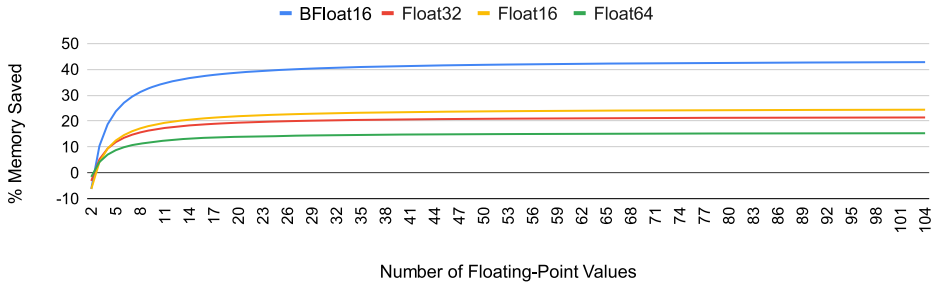


Fig. 7. Memory savings using exponent sharing in various precision formats. The maximum memory savings possible in BFloat16 is 43.75%, in Float32 is 21.875%, in Float16 is 25%, and in Float64 is 15.625%. The number of weights should be more than 2 for sharing exponents.

ALGORITHM 4: Exponent Share Aware Retraining

Input: Pre-Trained Model (M), number of epochs (E)
Result: Compressed Model

$k = log_{10}(2^{Mant}) - 1$ `// Mant = length(Mantissa of floating-point weights)`
$a = 0$ `// % Drop in accuracy`
Set memory savings (m_s) and accuracy drop (a_d) user-defined thresholds
for $layer \leftarrow 0$ **to** L **do**
  Share exponents using Algorithm 3
end
Estimate the memory saved by exponent sharing ($M_{savings}$)
if $M_{savings} <= m_s$ *or* $a > a_d$ **then**
  Exit
else
  for $epoch \leftarrow 0$ **to** E **do**
    for $layer \leftarrow 0$ **to** L **do**
      Change the number of digits after decimal in floating-point weights to k
    end
    Do training
  end
  Test the model and report a
  $k = k - 1$
  Go to step 4
end

The mantissa length of the floating-point standard restricts the compression achieved using exponent sharing. In BFloat16, the memory savings cannot exceed 43.75% because of a fixed length of mantissa (7 bits). Figure 7 shows the possible memory savings in different floating-point formats.

The worst case for exponent sharing is then when the exponent table has all possible distinct values Additionally, exponent sharing is not beneficial when the number of weights is less than 3, as it causes memory overhead to store the exponent table as shown in Figure 7.

3.2 Mantissa Approximation and ESART

The results of iterative weight approximation and its impact on the accuracy of different models are shown in Section 4.1.1. The results, the detailed discussion of which we postpone to a later

Table 1. Real Float v/s Float Value Stored in Memory

Real Float	Value Stored as Float	No. of Decimal Digits Matching between Real Float and Value Stored as Float
0.1987376154	0.1987376213	7
1.987376154	1.987376213	6
19.87376154	19.87376213	5
198.7376154	198.7376099	4
1987.376154	1987.376099	3
19873.76154	19873.76172	2
198737.6154	198737.6094	1

stage, show that the weight-approximated models that use exponent sharing save more memory with each iteration using Algorithm 1. However, the accuracy decreases as the number of iterations increases.

Apart from exponent-based weight approximations, we study weight approximation by restricting the length of fraction (equivalently the length of mantissa) in weights to a fixed number of decimal digits. This leads to changes in the range of weight values, which has the potential to enhance the benefits of exponent sharing. For example, if we consider the IEEE Float32 format, the mantissa is 23 bits long. The 23 bits of mantissa lead to 6.92 ($log_{10}2^{23}$) or ≈7 decimal digits of precision. If we consider a real value 0.198737615384998654, the value stored in the float will be 0.**1987376**21307373046875. Only the first 7 digits are stored as it is in memory. As the length of the integer part increases, the fractional part goes on decreasing as shown in Table 1.

We also applied mantissa (fraction) approximation to pre-trained models in an iterative manner where each iteration reduces the length of fraction in weights by 1 digit. We found that this improved the memory savings in models discussed in Section 4.1.1. However, few models lose accuracy with this approximation. For an example, the accuracy of ResNet18 drops to 83% from 91% with mantissa approximation. We discuss related results in detail in Section 4.1.3.

To reduce this accuracy loss, an ESART method is proposed, which is shown in Algorithm 4. This method is applied to the pre-trained models in the context of this article, and therefore it is called *retraining*. ESART aims to generate a model that will save the maximum possible memory after exponent sharing with the least or no impact on accuracy.

A pre-trained model is given as an input to Algorithm 4. The algorithm first calculates the number of fractional digits in decimal format from the IEEE floating-point format of the weights (line 1). Reducing mantissa length limits the range of floating-point weights and hence the exponent distribution. Therefore, every iteration (lines 4–20) of Algorithm 4 reduces the precision of decimal digits of weights by 1 digit (line 13). The training algorithm tries to minimize the error for the given precision of decimal digits in each iteration of retraining. It may be noted that we are not proposing any new training algorithm per se for loss minimization. Algorithm 4 uses the loss minimization training algorithm in any given machine learning framework like PyTorch [30] and TensorFlow [2], among others. The model is tested to check the accuracy impact during each iteration (line 17). This process repeats until the user-defined thresholds for memory savings or accuracy are met (lines 8 and 9). After the last successful iteration, the model will have the least possible distinct exponents per layer. We discuss the results in Section 4.

4 Experiments and Results

We have demonstrated all of the proposed model size reduction methods that are shown in Figure 3 on the pre-trained models taken from the work of Phan [32]. The models discussed are ResNet18

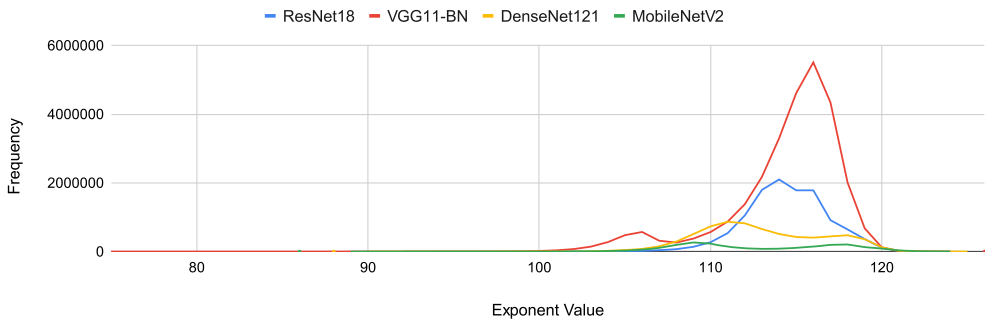


Fig. 8. Exponent frequency distribution in different models trained on CIFAR10.

[15], VGG11-BN [36], DenseNet121 [18], and MobileNetV2 [17]. The models are pre-trained on the CIFAR10 dataset [23] and have their test accuracy as 91%, 91%, 91%, and 90%, respectively. We performed our experiments on a Tesla T4 GPU from Google Colab [3]. The exponent frequency distributions in the weights of these models are as shown in Figure 8. It is important to note that the proposed methods can also be applied to any model in general, irrespective of whether the target hardware needs model compression or not. Hence, they can also be used with larger models.

For exponent sharing in a model, we make an exponent table per layer that holds distinct exponents in that layer. If there are e distinct exponents, then $i = \lceil log_2(e) \rceil$ index bits are required for referring it. During exponent sharing, the exponent field from the floating-point formats is replaced by an index field. In the BFloat16 floating-point format, the exponent field is half of its total length, whereas in the IEEE single-precision floating format, it is a quarter of its total length. Thus, the BFloat16 format gains more memory savings with exponent sharing. We used pre-trained models in BFloat16 format for the experiments related to Algorithm 1. For Algorithm 4, we used pre-trained models in IEEE single precision as input and converted the output model to BFloat16.

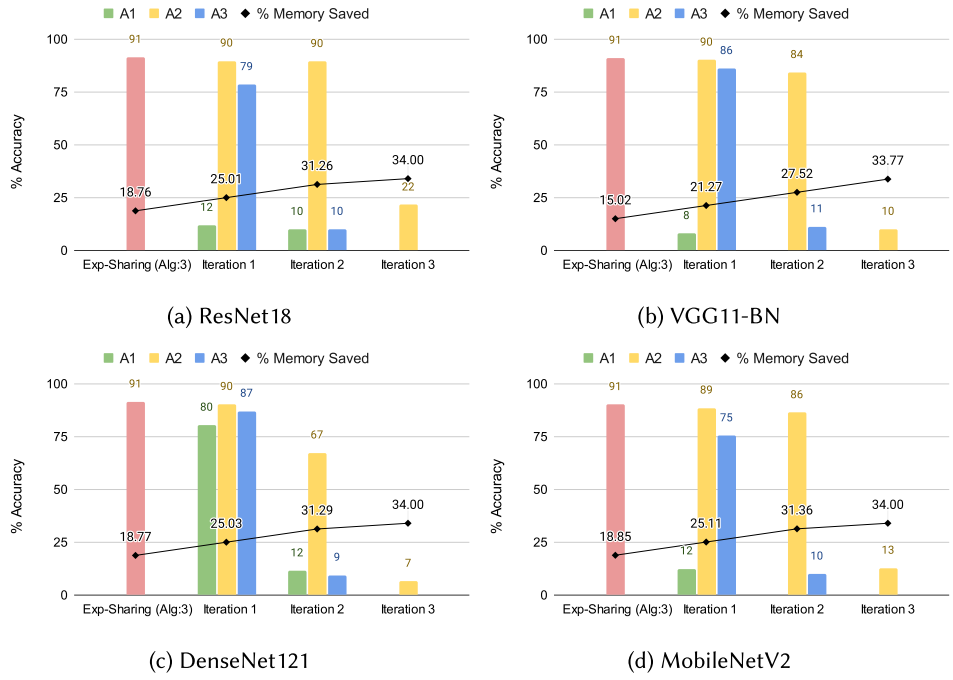
4.1 Approximation of Weights in Pre-Trained Models

4.1.1 Weight Approximation Using Exponent Magnitude. We have implemented Algorithm 1 of weight approximation on pre-trained models. This algorithm approximates the weights per layer in such a way that 1 bit less is required for indexing the exponent table. The iterations continue until either the compression reaches its maximum or the accuracy degradation exceeds 10% (line 8 of Algorithm 4: $m_s = 0$ or $a_d = 10\%$). Figure 9 shows the results of iterative weight approximation on different models. For each model, we show the impact of exponent sharing (Algorithm 3) on memory and accuracy followed by the impact of iterative weight approximations (Algorithm 1).

The different models have varying weights and sets of exponent values, and their distributions are also different. Consequently, the benefit of sharing exponents varies for each model. Figure 9 illustrates that exponent sharing on pre-trained models resulted in memory savings of around 15% to 19%, without affecting accuracy. Additionally, the outcomes of weight approximations carried out using the methods proposed in Section 3.1 are presented. These approximations were performed using one of the three substitutes (i.e., A1/A2/A3) at a time, as shown in line 12 of Algorithm 1.

In the case of substitute A1, which involves removing non-salient weights, the accuracy of all models drops by more than 10%. Therefore, none of the models qualify for a second iteration.

For A2 as the substitute, which involves replacing non-salient weights with the nearest salient weight in a layer, the accuracy of ResNet18, VGG11-BN, DenseNet121, and MobileNetV2 models



(a) ResNet18 (b) VGG11-BN

(c) DenseNet121 (d) MobileNetV2

Fig. 9. Impact of proposed iterative weight approximation methods (Algorithm 1) on the memory savings and accuracy of different models.

became 90%, 90%, 90%, and 89%, respectively. All models qualified for the second iteration, as the drop in accuracy was less than 1%. In the second iteration, the accuracy of ResNet18 remained the same, whereas the accuracy of VGG11-BN and MobileNetV2 models dropped by 7.7% and 4.4%, respectively, qualifying them for the third iteration. However, the accuracy of DenseNet121 dropped by more than 10%, which stopped the algorithm in the second iteration. Finally, in the third iteration, the accuracy of ResNet18, VGG11-BN, and MobileNetV2 models dropped by more than 10%, which stopped further weight approximation.

In the case of substitute A3 (i.e., adjusting the exponent and mantissa of non-salient weights), VGG11-BN and MobileNetV2 only qualified for the second iteration. However, in the second iteration, their accuracy dropped to more than 10% and the algorithm stopped. We ran the algorithm on a few models even after stopping criteria of Algorithm 1 was reached only to study the possible memory savings. Iteration 3 shows that accuracy degrades severely for these models with A1/A2/A3.

In each iteration, the proposed method saves approximately 3% more memory for any substitute, A1, A2, or A3. However, A2 showed the least accuracy drop when compared to A1 and A3. Since frequency of exponents also plays a role, A2 is considered for further investigation in Section 4.1.2 and comparison between its magnitude-based version and the frequency-based version.

4.1.2 Weight Approximation Using Exponent Frequencies. We conducted a study on weight approximations using exponent frequencies similar to exponent magnitude. In this case, the salient exponents ($2^{\lceil log_2(e) \rceil - 1}$) were those with higher frequencies rather than higher magnitudes. We used Algorithm 1 with weight adjustment without altering the important exponents (A2) for all non-salient weights. We observed that taking exponent frequencies into account for weight approximations has a greater impact on the accuracy of all models, as demonstrated in Table 2.

Table 2. Comparison of Weight Approximation by Method A2 Considering Exponent Magnitudes and Exponent Frequencies

Case	ResNet18				% Memory Saved
	By Exponent Magnitude		By Exponent Frequency		
	% Accuracy	% Weights Approximated	% Accuracy	% Weights Approximated	
Original	91.48	None	91.48	None	NA
Iteration 1	89.73	0.06	85.93	0.03	25.0
Iteration 2	89.5	15.28	80.95	2.85	31.25
Iteration 3	**21.59**	85.52	**10.4**	57.79	37.5
Case	**VGG11-BN**				**% Memory Saved**
	By Exponent Magnitude		**By Exponent Frequency**		
	% Accuracy	**% Weights Approximated**	**% Accuracy**	**% Weights Approximated**	
Original	91.07	None	91.07	None	NA
Iteration 1	90.17	0.20	88.80	0.37	21.27
Iteration 2	84.29	6.15	**42.69**	10.35	27.52
Iteration 3	**10.06**	42.86	NA	NA	33.77
Case	**DenseNet121**				**% Memory Saved**
	By Exponent Magnitude		**By Exponent Frequency**		
	% Accuracy	**% Weights Approximated**	**% Accuracy**	**% Weights Approximated**	
Original	91.49	None	91.49	None	NA
Iteration 1	90.29	1.61	**10**	0.53	25.02
Iteration 2	**67.34**	61.13	NA	NA	31.28
Case	**MobileNetV2**				**% Memory Saved**
	By Exponent Magnitude		**By Exponent Frequency**		
	% Accuracy	**% Weights Approximated**	**% Accuracy**	**% Weights Approximated**	
Original	90.22	None	90.22	None	NA
Iteration 1	88.59	4.44	**6.23**	1.38	25.1
Iteration 2	86.39	57.23	NA	NA	31.36
Iteration 3	**12.77**	78.86	NA	NA	37.6

4.1.3 Mantissa Approximation in Weights. The mantissa approximation, due to the properties shown in Table 1, is also done in an iterative manner. The first iteration starts by limiting the length of fraction of decimal weights to 6 digits. We continue the iterations until the accuracy drops by more than 10%. Each iteration narrows the range of real values, which reduces the distinctness in exponents. The results are presented in Figure 10 along with ESART. With each iteration, the precision of decimal digits in the weights of pre-trained models is reduced by 1. Restricting the number of decimal digits to 3 (i.e., 7-bit mantissa) does not harm the accuracy. This is nothing but a conversion of Float32 weights to BFloat16 [20]. Therefore, the accuracy of all models is not compromised until the fourth iteration. Beyond that, there is degradation of accuracy. We study ESART to reduce the accuracy loss.

4.2 Results of ESART

As discussed in Section 4.1, all proposed methods of weight approximation on the pre-trained models save memory, but they do so at the cost of accuracy loss. Training generates a new set of weights in each epoch, subsequently generating new exponents. It is crucial to minimize the

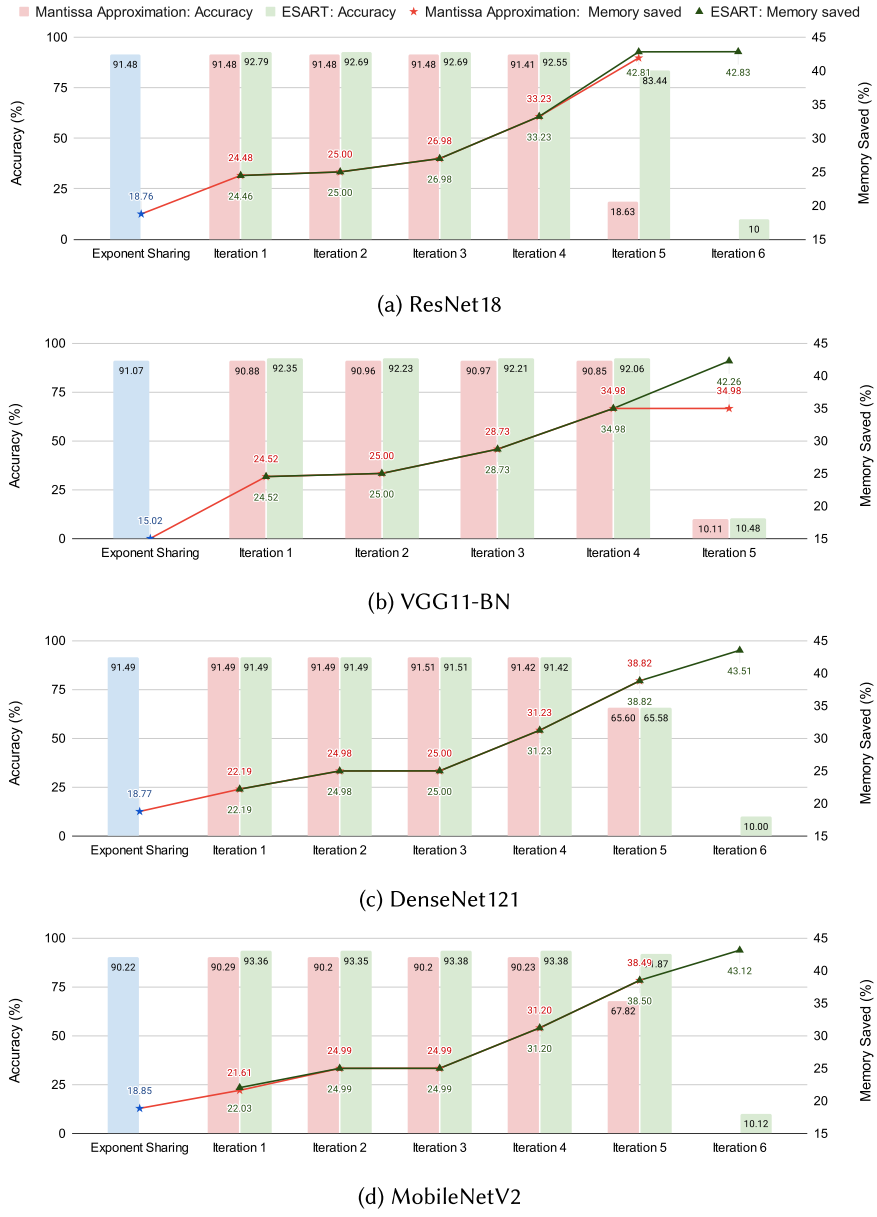
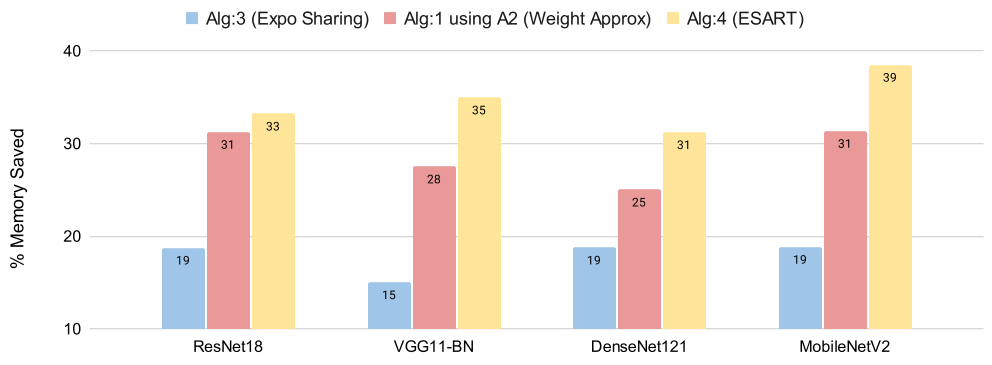


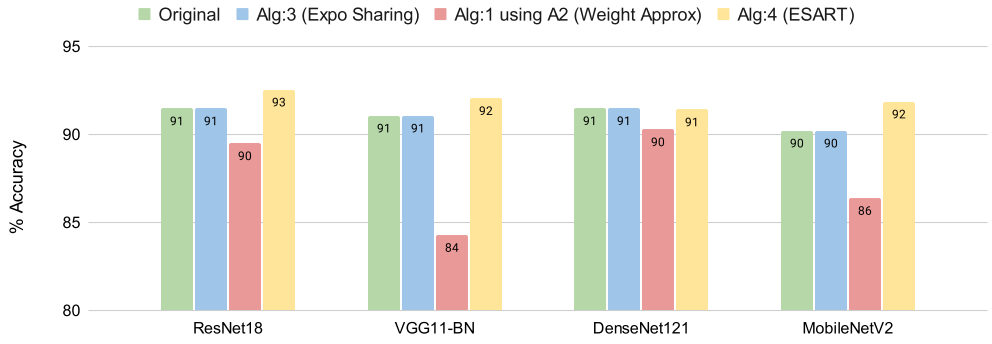
Fig. 10. Iterative mantissa approximation and ESART. Each successful iteration reduces the precision of decimal digits in weights by 1 digit (line 18 in Algorithm 4). The memory savings reported are for BFloat16 format as discussed in Section 4.2.

number of distinct exponents to maximize memory savings. However, using exponent-based weight approximations during training (i.e., A1/A2/A3 (Algorithm 2)) does not guarantee that the weights learned will have a reduced number of exponents. Comparatively, mantissa approximation showed higher memory savings.

In this section, we present the results of a study on improving accuracy through ESART proposed in Algorithm 4. We apply this algorithm on the model stored in Float32 format. After each



(a) Impact of proposed methods on memory savings



(b) Impact of proposed methods on model accuracy

Fig. 11. Comparison of "Best Results" from exponent and mantissa-based weight approximations studied in this article.

iteration of Algorithm 4, the precision was changed to BFloat16. The user-defined parameters in line 3 of Algorithm 4 were set to $m_s = 0$ (for memory savings) and $a_d = 10\%$ (for accuracy drop). The results are shown in Figure 10.

In the case of ResNet18, we saved 33.22% memory without any impact on accuracy. If a drop in accuracy of more than 10% is acceptable, it could save up to 42.8% of memory. Similarly, VGG11-BN saved 34.92% of memory without any impact on accuracy, whereas DenseNet121 saved 31.22% of memory without any loss of accuracy and up to 38.8% of memory with a 27% drop in accuracy. In the case of MobileNetV2, we observed a 38.5% memory savings without affecting accuracy. Additionally, this approach demonstrated an improvement in accuracy, which we attribute to retraining, in all models compared to the mantissa approximation (Section 4.1.3) while achieving the same or more memory savings. ESART still gives reasonable accuracy in iteration 5 (except for VGG11-BN): 83% for ResNet18, 65% for DenseNet121, and 90% for MobileNetV2 with 35% or more memory savings. However, only mantissa approximation is inferior to ESART in these iterations.

4.2.1 Comparison of Models Generated after the Proposed Exponent- and Mantissa-Based Weight Approximations. Among the proposed exponent based weight approximation methods (Algorithm 2), the weight approximation method that replaces non-salient weights with the nearest salient weight (i.e., A2-magnitude based) proved to be the best. Therefore, we compared the memory savings and accuracy of the models with weight approximation (Algorithm 1) with the models generated after ESART (Algorithm 4). The results are shown in Figure 11.

Table 3. Accuracy Comparison with Other Methods

Model	Accuracy (%)			
	Original	ESART (Alg:4)	Static Pruning [14]	Quantization [14]
ResNet18	91.48	93.00	62.30	91.40
VGG11-BN	91.00	92.00	64.01	90.52
DenseNet121	91.49	91.49	61.63	89.47
MobileNetV2	90.22	93.00	56.03	88.70

Table 4. Model Sizes in BFloat16 Format

Model	Model Size (MB)	
	Original	ESART
ResNet18	21.30	14.22
VGG11-BN	53.66	34.89
DenseNet121	13.19	9.07
MobileNetV2	4.23	2.60

We found that ESART (Algorithm 4) resulted in the better memory savings without accuracy loss in all models. Thus, we conclude that out of all algorithms studied in this article, ESART (Algorithm 4) achieved the highest memory savings without any loss in accuracy compared to the input pre-trained model.

4.3 Comparison with Other Model Compression Methods

We compared the model generated by ESART with the models generated by other model compression methods like pruning and quantization from PyTorch [30]. We observe the impact of these methods on the following different factors.

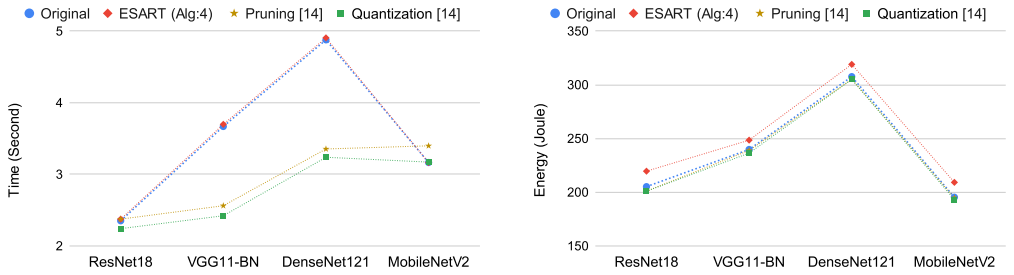
4.3.1 Accuracy. The accuracy remains unchanged after ESART up to four iterations, as shown in Figure 10. It even improves the accuracy slightly, as retraining is done on the top of a pre-trained model. In the case of pruning, the accuracy drops beyond 30%. However, quantization shows less than a 1% accuracy drop. Table 3 shows the accuracy loss in different cases.

4.3.2 Memory Savings. When only exponent sharing [22] is applied on the models, then memory savings of 18.75%, 15.02%, 18.77%, and 18.84% are observed in ResNet18, VGG11-BN, DenseNet121, and MobileNetV2, respectively. However, the memory savings are significantly improved by ESART. The memory savings become 33.22%, 34.97%, 31.22%, and 38.48% in ResNet18, VGG11-BN, DenseNet121, and MobileNetV2, respectively, without any accuracy loss. The difference in the size of the new compressed models is shown in Table 4. Higher memory savings are also possible with some loss in accuracy, as shown in Figure 10 (refer to iteration 5).

For comparison with pruning, the pre-trained models were pruned to save memory equivalent to that saved by models after ESART. For example, ResNet18 was pruned by 33.22% and VGG11-BN by 34.97%. Quantization saved 50% memory in all models, as it changed the precision of weights from BFloat16 to Int8. However, as discussed in Section 4.3.1, it resulted in a loss of some accuracy.

4.4 Execution Time and Energy at Inference

In our experiments with PyTorch (in Google Colab), we stored the weight values per layer into the proposed floating-point storage format. This format requires different components such as sign, index, mantissa, and also an exponent table, as shown in Figure 6. When our method is not applied, a pre-trained model requires a single read to access the floating-point weights. With



(a) Inference Time (b) Energy

Fig. 12. Comparison of inference time and energy consumed during inference (batch size 256) in models generated after different model compression methods.

Table 5. Execution Time of ResNet18 Inference of a Single Image

Name	Pre-Trained		ESART Processed Model	
	Original	Cache Flush	Original	Cache Flush
Self CPU time total	8.224 ms	11.648 ms	12.706 ms	13.008 ms
Self CUDA time total	4.157 ms	4.157 ms	7.450 ms	7.459 ms

the proposed format, each layer of a model now reads separately stored components. Indices are used to retrieve exponents from the exponent table. We used the function "torch.ldexp(mantissa, exponent)" from PyTorch to transcode individual reads into the original floating-point weight. In the upcoming sections, we compare the time and energy required by a batch of input images during inference. During testing, we compared the time required for inference in two cases: (1) when it is a normal pre-trained model, and (2) when such exponent sharing is implemented. We ran this simulation 15 times for both cases and took the average to show the execution impact.

4.4.1 Inference Time. The proposed ESART technique takes slightly more time for inference, as shown in Figure 12(a), but it is still comparable to the original (within 1%) (i.e., pre-trained model). This technique ensures that there is no or very little loss in accuracy, which is not the case for models generated after pruning and quantization. As illustrated in Figure 6, exponent sharing requires three additional reads to form a floating-point number from its sign, index, and mantissa. During sequential execution on processors, the execution time is affected by exponent sharing. We implement exponent sharing on top of model generated after ESART. Therefore, there is an overhead in execution cycles in our experiments. The impact shown here is without any software optimization. Indeed, we explicitly cleared the cache memory to observe the impact on inference time when the reads are happening from memory every time. If explicit cache flushing is not done, then there is some advantage in terms of inference time. For example, consider the inference of ResNet18 with and without caching. There is approximately a 2.35% reduction in execution time when caching is utilized in PyTorch, as shown in Table 5. These execution times are obtained using PyTorch Profiler. On detailed profiling, one can see that the overhead is mainly caused by functions *torch.ldexp()* and *pow()*, which are used to transcode the compressed components of weights back to the standard floating-point formats.

4.4.2 Energy Consumption. Due to the greater number of reads happening after exponent sharing, the energy consumption is slightly increased (within 5%). Figure 12(b) shows the results on

Table 6. Time Overhead for ESART in FP32

Model	Time (Minutes)			Precision of Decimal Digits (k)
	Training (100 Epochs)	ESART (10 Epochs)	Overhead (%)	
ResNet18	17.733	2.619	14.772	2
VGG11-BN	14.717	2.057	13.978	3
DenseNet121	42.876	9.07	21.154	3
MobileNetV2	35.067	7.513	21.425	2

energy consumption by models compressed after different compression methods. We used the CodeCarbon [35] framework to analyze the energy consumption in PyTorch. Currently, we are calculating the memory savings from proposed weight approximations in PyTorch. The underlying bitwidths for each field are still the standard ones (Float32). Therefore, the impact on energy consumption does not account for variable bit lengths for different components in the proposed storage format, in the case of processors. A detailed optimized software implementation that would exploit parallel reads or reads overlapping with computation would definitely reduce the overhead in processors as well. Yet, we will consider such an optimized implementation for future work, as it would require exploiting processor characteristics for memory coalescing and so forth. However, Section 4.5 presents the implementation of LeNet on FPGA using the proposed ESART method where we observe both a reduction in overall power dissipation and appropriate memory savings.

4.4.3 Execution Overhead of ESART. The training time overhead of ESART on different models is shown in Table 6. The time for ESART is recorded for the smallest value of k (line 1 of Algorithm 4) that does not affect the accuracy of the model. We report the overhead for 10 epochs because when the batch size is 256, there is only minor variation in accuracy up to 10 epochs, as shown in Figure 13, which displays the changes in accuracy and execution time during each epoch of ESART.

4.5 Results on FPGA

The proposed exponent sharing method departs from standard floating-point storage formats by employing a variable-length index field, unlike the fixed-size exponent field in standard floating-point formats. This necessitates specialized hardware that can efficiently handle this variability on a per-layer basis. In our previous work [21], we explored the implementation of exponent sharing on FPGAs due to their ability to perform parallel reads and accommodate variable bitwidths. We proposed a model for statically analyzing the impact of exponent sharing on clock cycles and introduced a specialized hardware structure (depicted in Figure 14). Experiments using Vivado tools demonstrated a negligible impact on clock cycles across all tested layers of various models, with the maximum observed increase being less than 1% when parallel reading happens.

In this section, we present the results of applying ESART on a LeNet model trained on the MNIST dataset, which initially had an accuracy of 97.64% and is stored in Float32 floating-point standard. The model achieved a 15% reduction in memory usage with 97.16% accuracy in the fifth iteration of ESART (the sixth iteration encountered the stopping criteria of ESART, with accuracy degrading to 10%). Table 7 illustrates the impact on factors such as clock cycles, power, and resources after implementing ESART. The model, compressed by ESART, is implemented using pipelining (with an initiation interval of 1), which further reduces the impact on clock cycles through parallel reads. However, unlike on CPU, the power impact is less after ESART. These results were obtained using Vivado tools, targeting the Zynq SoC FPGA on Zed Board [1] as the underlying hardware.

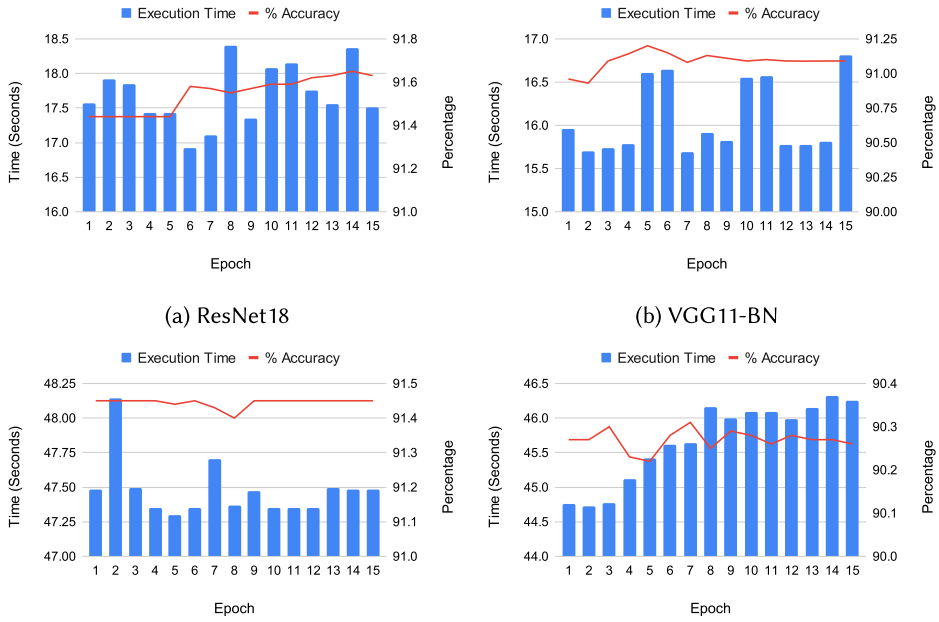


Fig. 13. Study of accuracy and execution time in each epoch during a single iteration of ESART.

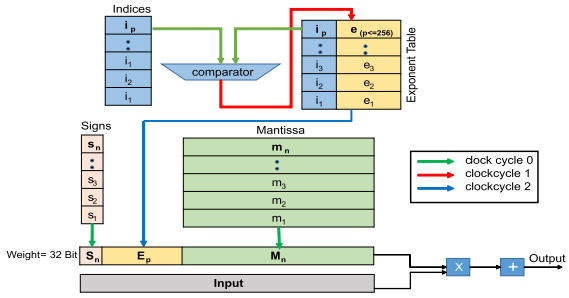


Fig. 14. Hardware architecture diagram for transcoding and exponent sharing (applicable to floating-point formats like IEEE Float32 format) [21].

Table 7. Performance of LeNet on FPGA

Name		Original	ESART with Pipeline
Clock Cycles		6,090,756	5,336,520
Power (W)		0.249	0.24
Resource Utilization on Zynq	BRAM_18K	29	28
	DSP48E	25	28
	FF	6,447	6,277
	LUT	5,693	5,866

5 Conclusion and Future Work

We proposed different methods to approximate weights and a retraining method that uses weight approximation, to improve the memory savings with exponent sharing without any loss in accuracy. Although results were presented on models that would not be considered large, the proposed methods can be easily applied to large models as well. Figure 3 shows the various methods proposed. We presented a systematic study of these methods to identify a performance-compression tradeoff. Among exponent-based weight approximation, the method that approximates weights by changing non-salient weights to the nearest higher salient weight (Section 4.1.1: A2) results in more memory savings compared to other proposed methods (A1/A2). The accuracy loss is also relatively less using A2. The proposed ESART (Algorithm 4) on top of a trained model saves more memory without any loss in accuracy. This retraining restricts the range of weights in a layer, which in turn reduces the distinct exponent but increases their frequency, allowing for greater sharing of exponents.

Exponent sharing (Algorithm 3) is a part of ESART (Algorithm 4) in this work. We compared models generated using ESART with the models generated by other state-of-the-art methods of model compression in terms of accuracy, inference time, and energy. There is some time and energy overhead, as exponent sharing adds three more reads for every weight. However, this overhead can be acceptable with other methods, as the proposed method preserves the accuracy of models. Moreover, this overhead can be reduced by software optimization methods like memory coalescing [7, 19] and on parallel hardware like FPGAs. During ESART, we currently limit the number of decimal digits in the weights to a fixed length in all layers. In our future research, we plan to investigate the optimal precision of decimal digits for each layer, aiming to minimize the impact on accuracy while maximizing memory savings.

It may be noted that knowledge distillation, as one other popular method for model compression, requires defining our own smaller student model, and during transfer learning, the training of this student model happens. There can be various architectures of student models that can undergo knowledge distillation before a model is finalized. This creates a large design space to explore to get a properly trained student model. It is thus clear that knowledge distillation, while it leads to smaller models, is orthogonal in approach to our proposed methods and a fair comparison would not be possible. To be more specific, when LeNet undergoes compression through weight quantization or static pruning in PyTorch, the model's structure remains largely unchanged. Only the weights in the model differ after compression. However, when knowledge distillation, as proposed in the work of Lopes et al. [26], is applied, layers of the model are removed, resulting in a reduction of the model size. It then needs to be retrained using the knowledge obtained from the original pretrained LeNet. Our methods can be applied once a suitable student model is available for further reduction in model size.

References

[1] Xilinx. 2018. Zynq-7000 SoC Data Sheet: Overview. Retrieved February 28, 2023 from https://www.mouser.com/datasheet/2/903/ds190-Zynq-7000-Overview-1595492.pdf

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Tiago Carneiro, Raul Victor Medeiros Da NóBrega, Thiago Nepomuceno, Gui-Bin Bian, Victor Hugo C. De Albuquerque, and Pedro Pedrosa Rebouças Filho. 2018. Performance analysis of Google Colaboratory as a tool for accelerating deep learning applications. *IEEE Access* 6 (2018), 61677–61685. https://doi.org/10.1109/ACCESS.2018.2874767

[4] Miguel A. Carreira-Perpinán and Yerlan Idelbayev. 2018. "Learning-compression" algorithms for neural net pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8532–8541.
[5] Sung-En Chang, Yanyu Li, Mengshu Sun, Runbin Shi, Hayden K.-H. So, Xuehai Qian, Yanzhi Wang, and Xue Lin. 2021. Mix and Match: A novel FPGA-centric deep neural network quantization framework. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*. IEEE, 208–220.
[6] Hsu-Hsun Chin, Ren-Song Tsay, and Hsin-I. Wu. 2021. A high-performance adaptive quantization approach for edge CNN applications. *arXiv:2107.08382 [cs.CV]* (2021).
[7] Jack W. Davidson and Sanjay Jinturkar. 1994. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/178243.178259
[8] Etienne Dupuis, David Novo, Ian O'Connor, and Alberto Bosio. 2020. On the automatic exploration of weight sharing for deep neural network compression. In *Proceedings of the 2020 Design, Automation, and Test in Europe Conference and Exhibition (DATE '20)*. IEEE, 1319–1322.
[9] Etienne Dupuis, David Novo, Ian O'Connor, and Alberto Bosio. 2020. Sensitivity analysis and compression opportunities in DNNs using weight sharing. In *Proceedings of the 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '20)*. IEEE, 1–6.
[10] Bryn Elesedy, Varun Kanade, and Yee Whye Teh. 2020. Lottery tickets in linear models: An analysis of iterative magnitude pruning. *arXiv preprint arXiv:2007.08243* (2020).
[11] Yoonhee Gil, Jong-Hyeok Park, Jongchan Baek, and Soohee Han. 2021. Quantization-aware pruning criterion for industrial applications. *IEEE Transactions on Industrial Electronics* 69, 3 (2021), 3203–3213.
[12] Jinyang Guo, Weichen Zhang, Wanli Ouyang, and Dong Xu. 2020. Model compression using progressive channel pruning. *IEEE Transactions on Circuits and Systems for Video Technology* 31, 3 (2020), 1114–1124.
[13] Manish Gupta and Puneet Agrawal. 2022. Compression of deep learning models for text: A survey. *ACM Transactions on Knowledge Discovery from Data* 16, 4 (Jan. 2022), Article 61, 55 pages. https://doi.org/10.1145/3487045
[14] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.). Vol. 28. Curran Associates, 1–9. https://proceedings.neurips.cc/paper_files/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf
[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv:1512.03385 [cs.CV]* (2015).
[16] Weixin Hong, Tong Chen, Ming Lu, Shiliang Pu, and Zhan Ma. 2020. Efficient neural image decoding via fixed-point inference. *IEEE Transactions on Circuits and Systems for Video Technology* 31, 9 (2020), 3618–3630.
[17] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tang, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.
[18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Densely connected convolutional networks. *arXiv:1608.06993 [cs.CV]* (2018).
[19] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 105–118. https://doi.org/10.1109/TPDS.2010.107
[20] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A study of BFLOAT16 for deep learning training. *arXiv:1905.12322 [cs.LG]* (2019).
[21] Prachi Kashikar, Olivier Sentieys, and Sharad Sinha. 2023. Lossless neural network model compression through exponent sharing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31, 11 (2023), 1816–1825. https://doi.org/10.1109/TVLSI.2023.3307607
[22] Prachi Kashikar, Sharad Sinha, and Ajeet Kumar Verma. 2021. Exploiting weight statistics for compressed neural network implementation on hardware. In *Proceedings of the 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS '21)*. IEEE, 1–4.
[23] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.
[24] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073* (2017).
[25] Jing Liu, Bohan Zhuang, Zhuangwei Zhuang, Yong Guo, Junzhou Huang, Jinhui Zhu, and Mingkui Tan. 2022. Discrimination-aware network pruning for deep model compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4035–4051. https://doi.org/10.1109/TPAMI.2021.3066410

[26] Raphael Gontijo Lopes, Stefano Fenu, and Thad Starner. 2017. Data-free knowledge distillation for deep neural networks. *arXiv preprint arXiv:1710.07535* (2017).
[27] Sashank Macha, Om Oza, Alex Escott, Francesco Caliva, Robbie Armitano, Santosh Kumar Cheekatmalla, Sree Hari Krishnan Parthasarathi, and Yuzong Liu. 2023. Fixed-point quantization aware training for on-device keyword-spotting. *arXiv:2303.02284 [eess.AS]* (2023).
[28] Fanxu Meng, Hao Cheng, Ke Li, Huixiang Luo, Xiaowei Guo, Guangming Lu, and Xing Sun. 2020. Pruning filter in filter. *Advances in Neural Information Processing Systems* 33 (2020), 17629–17640.
[29] Yuriy Mishchenko, Yusuf Goren, Ming Sun, Chris Beauchene, Spyros Matsoukas, Oleg Rybakov, and Shiv Naga Prasad Vitaladevuni. 2019. Low-bit quantization and quantization-aware training for small-footprint keyword spotting. In *Proceedings of the 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA '19)*. 706–711. https://doi.org/10.1109/ICMLA.2019.00127
[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. *arXiv:1912.01703 [cs.LG]* (2019).
[31] Zoran Peric, Milan Savic, Milan Dincic, Nikola Vucic, Danijel Djosic, and Srdjan Milosavljevic. 2021. Floating point and fixed point 32-bits quantizers for quantization of weights of neural networks. In *Proceedings of the 2021 12th International Symposium on Advanced Topics in Electrical Engineering (ATEE '21)*. 1–4. https://doi.org/10.1109/ATEE52255.2021.9425265
[32] Huy Phan. 2021. huyvnphan/PyTorch_CIFAR10. Retrieved August 13, 2024 from https://doi.org/10.5281/zenodo.4431043
[33] Masuma Akter Rumi, Xiaolong Ma, Yanzhi Wang, and Peng Jiang. 2020. Accelerating sparse CNN inference on GPUs with performance-aware weight pruning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 267–278.
[34] Ratshih Sayed, Haytham Azmi, Heba Shawkey, A. H. Khalil, and Mohamed Refky. 2023. A systematic literature review on binary neural networks. *IEEE Access* 11 (2023), 27546–27578. https://doi.org/10.1109/ACCESS.2023.3258360
[35] Victor Schmidt, Kamal Goyal, Aditya Joshi, Boris Feld, Liam Conell, Nikolas Laskaris, Doug Blank, Jonathan Wilson, Sorelle Friedler, and Sasha Luccioni. 2021. CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing. Retrieved August 13, 2024 from https://codecarbon.io
[36] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556 [cs.CV]* (2015).
[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.
[38] Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008* (2017).
[39] Paola Vitolo, Rosalba Liguori, Luigi Di Benedetto, Alfredo Rubino, Danilo Pau, and Gian Domenico Licciardo. 2023. A new NN-based approach to in-sensor PDM-to-PCM conversion for ultra TinyML KWS. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 4 (2023), 1595–1599. https://doi.org/10.1109/TCSII.2022.3224022
[40] Erwei Wang, James J. Davis, Daniele Moro, Piotr Zielinski, Jia Jie Lim, Claudionor Coelho, Satrajit Chatterjee, Peter Y. K. Cheung, and George A. Constantinides. 2023. Enabling binary neural network training on the edge. *ACM Transactions on Embedded Computing Systems* 22, 6 (Nov. 2023), Article 105, 19 pages. https://doi.org/10.1145/3626100
[41] Ziwei Wang, Martin A. Trefzer, Simon J. Bale, and Andy M. Tyrrell. 2021. Adaptive integer quantisation for convolutional neural networks through evolutionary algorithms. In *Proceedings of the 2021 IEEE Symposium Series on Computational Intelligence (SSCI '21)*. IEEE, 1–7.
[42] Yanguang Xu, Jianwei Sun, Yang Han, Shuaijiang Zhao, Chaoyang Mei, Tingwei Guo, Shuran Zhou, Chuandong Xie, Wei Zou, and Xiangang Li. 2022. Audio-visual wake word spotting system for MISP Challenge 2021. In *Proceedings of the 2022 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '21)*. 9246–9250. https://doi.org/10.1109/ICASSP43922.2022.9746762
[43] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5687–5695.
[44] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).

Received 23 February 2024; revised 6 July 2024; accepted 1 August 2024