ReSprop: Reuse Sparsified Backpropagation

Negar Goli, Tor M. Aamodt
University of British Columbia
Vancouver, Canada
{negargoli93,aamodt}@ece.ubc.ca

Abstract

The success of Convolutional Neural Networks (CNNs) in various applications is accompanied by a significant increase in computation and training time. In this work, we focus on accelerating training by observing that about 90% of gradients are reusable during training. Leveraging this observation, we propose a new algorithm, Reuse-Sparse-Backprop (ReSprop), as a method to sparsify gradient vectors during CNN training. ReSprop maintains state-of-the-art accuracy on CIFAR-10, CIFAR-100, and ImageNet datasets with less than 1.1% *accuracy loss while enabling a reduction in back-propagation computations by a factor of* $10\times$ *resulting in a* $2.7\times$ *overall speedup in training. As the computation reduction introduced by ReSprop is accomplished by introducing fine-grained sparsity that reduces computation efficiency on GPUs, we introduce a generic sparse convolution neural network accelerator (GSCN), which is designed to accelerate sparse back-propagation convolutions. When combined with ReSprop, GSCN achieves* $8.0\times$ *and* $7.2\times$ *speedup in the backward pass on ResNet34 and VGG16 versus a GTX 1080 Ti GPU.*

1. Introduction

Convolutional neural networks (CNNs) have been tremendously successful in many modern machine learning applications [8, 15, 29, 53, 56, 57]. Prior work has adopted two main strategies to accelerate CNN training: (1) reducing the number of iterations per compute node required to converge using techniques such as batch normalization [23], parallelize training with data or model parallelism [10, 31], and importance sampling [27, 28]; (2) reducing the amount of computation per iteration using techniques such as stochastic depth to remove layers during training [22], randomized hashing to reduce the number of multiplications [54], quantization [6, 58, 62] and sparse training [13, 35, 55, 59]. We explore the second strategy and propose Reuse Sparse Backprop (ReSprop), a novel way to sparsify convolution computations during training[1].

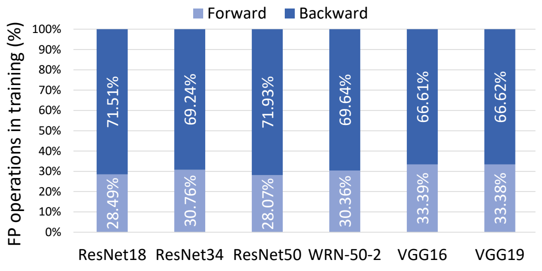


Figure 1. Percentage of floating point operations during backward and forward pass in training different architectures.

Sparse convolutions decrease computational cost by reducing the number of multiplication and addition operations. Recent related work [20, 37, 38, 40, 42] studies different approaches to sparsifying inference, and many studies [1, 2, 9, 16, 46] have designed accelerators to exploit sparsity in inference; however, there is limited work on sparse training [13, 35, 44, 55, 59]. Our measurements shown in Figure 1 indicate that back-propagation consumes around 70% of the time during training. MeProp [55, 59] and DSG [35] accelerate back-propagation convolutions using different sparsification methods. However, we observe that meProp fails to converge while training deeper networks or when using large datasets (Sections 3 and 5) and DSG loses more accuracy and achieves less training speedup compared to ReSprop. ReSprop reduces the computation overhead of back-propagation by reusing gradients to sparsify back-propagation convolutions. ReSprop overcomes these limitations and can be accelerated by hardware similar to recently proposed inference accelerators [1, 2, 9, 16, 46].

Our observations (in Section 3.2) demonstrate that updating a small portion of the gradient components each iteration and replacing the rest with the previous iteration's gradient component values is sufficient for maintaining state-of-the-art accuracy. The ReSprop algorithm (Section 4) exploits gradient reusability and sparsifies the gradients in the

[1]Source code available at https://github.com/negargoli/ReSprop

back-propagation convolutions up to 90% with less than 1.1% loss of accuracy on the ImageNet dataset. ReSprop has less than 2% computation overhead and less than 16% memory footprint overhead while training the ImageNet dataset with batch sizes larger than 128. For 90% sparsity, we calculate ReSprop theoretical speedups between 9.3× and 9.8× for backward pass calculations and, as a consequence of Amdahl's Law [4], between 2.5× and 2.9× for the overall training process on different architectures. Moreover, we estimate the speedup that might be achieved on a custom hardware accelerator. Specifically, we propose a novel *Generic Sparse Convolutional Neural network (GSCN)* accelerator hardware architecture (Section 5.4). GSCN is designed to accelerate sparse back-propagation convolutions and based on SCNN [46], an accelerator proposed by NVIDIA for sparse convolutions. Our results (Section 5.4) show ReSprop on GSCN achieves 8.0× speedups versus a GPU on the backward pass of ResNet34.

2. Related Work

Dense to sparse networks by weight pruning: Creating sparse networks by eliminating the weights has an extensive history. LeCun *et al.* [33]; Karnin [26]; Hassibi and Stork [18] present the early work of network pruning using second-order derivatives as the pruning criterion. Han *et al.* [17] propose parameter magnitude as the pruning criterion and introduced the pipeline of training, pruning, and fine-tuning. There are also pruning methods with different pruning criteria [37, 38, 42] and approaches to removing channels and filters [20, 34, 36, 40, 43, 60]. These often involve a re-training phase, which, contrary to our motivation, increases training time.

Sparse training: More recent studies try to find the sparse network during training through a prune, redistribute, and regrowth cycle. Bellec *et al.* [7]; Mocanu *et al.* [41]; Mostafa and Wang *et al.* [44] propose different regrowth methods for sparsifying the networks through training. Dettmers *et al.* [13] present faster training by sparse momentum, which uses the exponentially smoothed gradients as the criterion for pruning and regrowth weights. A different approach to accelerate training is sparsifying activations. Liu *et al.* [35] introduce a dynamic sparse graph (DSG) structure, which activates only a small amount of neurons at each iteration via a dimension-reduction and accelerates forward and backward passes. Methods that maintain sparse gradients throughout training are most closely related to our work. Sun *et al.* [55] and Wei *et al.* [59] introduce meProp, an algorithm which targets computation reduction in training by sparsifying gradients. They demonstrate meProp convergence while training a network with two convolutional layers on the MNIST dataset at 95% gradient sparsity. However, they do not analyze larger datasets and deeper networks.

Reuse gradients: Principle stochastic variance reduction (SVR) techniques including SVRG [24], SAGA [11], and their variants reuse the gradients for updating the weights on smooth strongly-convex optimization problems. Recent works explore the extension of SVR approaches to general non-convex problems [3, 50]. However, the faster theoretical convergence rate of the SVR methods is not a guarantee of better empirical performance in deep neural networks [12]. ReSprop reuses gradients in a different way than SVR. ReSprop reuses gradients between successive mini-batches to sparsify back-propagation calculations. The goal of ReSprop is reducing computation, not variance. We show that our method reaches state-of-the-art accuracy with minimal loss while having 10× computation reduction in back-propagation for different network architectures with varying widths and depths.

3. Gradient reuse

In this section, we discuss the motivation behind ReSprop. We compare reusing gradients against sparsifying gradients (meProp).

3.1. Notation and Preliminaries

Convolution is the predominant operation in CNNs. The output of the l^{th} layer in the CNNs' forward-propagation is obtained by:

$$y_{l+1} = w_l \otimes a_l \tag{1}$$

Where a_l and w_l denote activations and weights at layer l, respectively, and $\otimes$ is the convolution operation. In back-propagation the l^{th} layer receives the *output gradient* of the $l+1^{th}$ layer. The output gradient is the gradient of the loss (L) with respect to the layer's output ($\frac{\partial L}{\partial y_{l+1}}$). The output gradient is used to compute the gradient of input activation ($\frac{\partial L}{\partial a_l}$) and the gradient of weights ($\frac{\partial L}{\partial w_l}$). The back-propagation convolutions for calculating gradient of inputs and weights at the l^{th} layer can be defined as [32]:

$$\frac{\partial L}{\partial a_l} = \frac{\partial L}{\partial y_{l+1}} \otimes w_l^t \tag{2}$$

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial y_{l+1}} \otimes a_l^t \tag{3}$$

Mini-batch training applies the above equations and updates the model parameters at each iteration (mini-batch). In this study, mini-batch training allows us to leverage the correlation among output gradient components of consecutive iterations and facilitates reusing the output gradient components. We use "gradients" to refer to individual components of the gradient vector throughout the paper.

3.2. Approach and Key Insights

Our approach to accelerate training is to modify back-propagation calculations. The *output gradient* vector and in turn the vectors dependant on it (Eq. 2 and 3) are updated in the backward pass. In essence, ReSprop precalculates a portion of the output gradient vector, and this, in turn, enables precomputing a portion of the backpropagated values. We conjectured that there are a large number of similar features between training samples, and this motivated us to explore reusing the output gradients among mini-batches. We focus on the feasibility of reusing a subset of the output gradients between consecutive iterations and measure the inter-iteration similarity of output gradients. We propose a reuse strategy to leverage precalculated output gradients from the previous iteration while performing computation only for significantly changed output gradients in the current iteration (mini-batch). We define our reuse strategy as follows: If a component of an output gradient compared to its previous iteration changes more than an adaptive threshold then we use the current (i^{th}) iteration value; otherwise, we reuse the value of the previous iteration. We introduce a vector we call the *hybrid output gradient (HG)*. We define HG such that it contains $x\%$ of the previous iteration's gradients and $(100-x)\%$ of the current iteration's gradients. Here, $x\%$ is called the reuse percentage. The HG for layer l at iteration i is defined as:

$$(HG_l)_i = (\frac{\partial L}{\partial y_{l+1}})_{i-1} + Th_l[(\frac{\partial L}{\partial y_{l+1}})_i - (\frac{\partial L}{\partial y_{l+1}})_{i-1}] \quad (4)$$

We use the notation $(a_l)_i$ to denote the value of vector a at layer l and iteration i. Each layer has its own adaptively adjusted threshold (T_l), which satisfies the reuse percentage. The function $Th_l(V)$, where Th stands for "Threshold", at layer l applied to output gradient vector V is defined as:

$$\forall v_i \in V: \; u_i = \begin{cases} v_i & |v_i| > T_l \\ 0 & |v_i| \leq T_l \end{cases} \quad (5)$$

where u_i represents the elements of output vector $Th_l(V)$ and T_l is a per layer adaptive threshold. In Section 4, we explain how to use $(HG_l)_i$ to sparsify back propagation using ReSprop. Here, we empirically show that HG_l is a good approximation to the original output gradient ($\frac{\partial L}{\partial y_{l+1}}$), and that it is feasible to train the network with the HG vector. To study the correlation between HG and the original output gradient, we investigate the angle preservation using cosine similarity. According to hyperdimensional computing theory [25], two independent isotropic vectors picked randomly from a high dimensional space d, are approximately orthogonal. If there is no correlation between the HG vector and the original output gradient, they would make an angle of approximately 90°. On the other hand, Anderson *et al.* [5] show that binarizing a random vector in high dimensional space d ($d \to \infty$), preserves the vector direction with minimal changes, and a random vector and its binarized version form an angle of around 37°. According to Anderson et al.'s observations, in a high dimensional space 37° is a relatively small angle between two vectors, so that both vectors have similar directions.

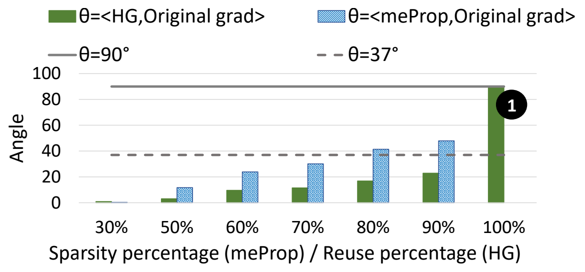


Figure 2. HG and meProp angles for different reuse percentages and sparsities, respectively. The angle is calculated by finding the average angle of all layers while training ResNet-18 on CIFAR-10 for 100 iterations (batch size=128).

Reuse	HG Val Acc	Sparsity	meProp Val Acc
50%	84.21 ± 0.09	50%	84.14 ± 0.08
60%	84.11 ± 0.06	60%	64.29 ± 0.07
70%	83.87 ± 0.10	70%	50.65 ± 0.13
80%	78.40 ± 0.14	80%	41.67 ± 0.25
90%	73.14 ± 0.17	90%	23.67 ± 0.23

Table 1. Validation accuracy of meProp and reuse strategy (HG) with different sparsities and reuse percentages, repectively. Training ResNet-18 on CIFAR-10 for **30 epochs** (batch size = 128, lr = 0.1 and optimizer = SGD).

Figure 2 demonstrates the angle between the original output gradient vector and both the HG vector (dark green bar) and meProp gradient (light blue bar). As shown at ➊, the angle between output gradient vectors of consecutive iterations is close to 90°. This indicates that successive output gradients are approximately orthogonal. However, we observe that reusing a subset of output gradient in consecutive iterations, via HG reuse strategy, reduces the angle between the original output gradients and the HG vector to less than 37°. We compare this strategy with meProp [55] by studying the angle preservation property and the validation accuracy of these algorithms. The meProp algorithm sets output gradients not ranked in the Top-K by magnitude to zero and calculates Eq. 3 and 2 with the sparse output gradient. Figure 2 shows the angle between the original output gradient and meProp. Since cosine similarity is undefined for a zero vector, the angle for 100% sparse meProp is not presented. We can see HG preserves the original output gradient direction far better than meProp's sparse output gradient. Table 1 further verifies the network convergence while reusing gradients. This table shows the validation accuracy of reusing output gradients with small magnitude change (HG Val Acc) compared to setting small magnitude

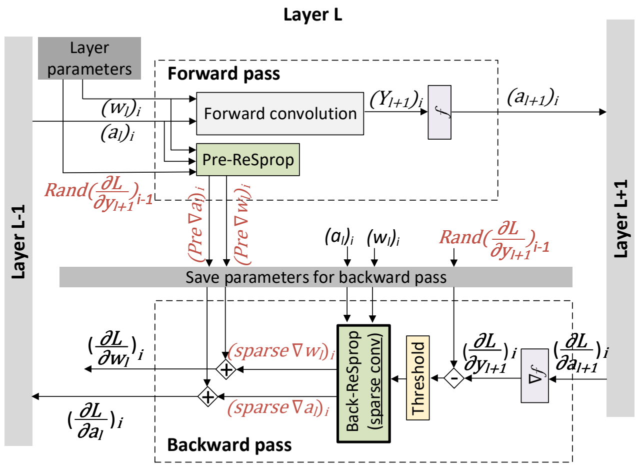

Layer L
Layer parameters
Forward pass
$(w_l)_i$
$(a_l)_i$
Forward convolution
$(Y_{l+1})_i$
f
$(a_{l+1})_i$
Pre-ReSprop
$Rand(\frac{\partial L}{\partial y_{l+1}})_{i-1}$
$(Pre\nabla a_l)_i$
$(Pre\nabla w_l)_i$
$(a_l)_i$ $(w_l)_i$
$Rand(\frac{\partial L}{\partial y_{l+1}})_{i-1}$
Save parameters for backward pass
Layer L-1
Layer L+1
$(\frac{\partial L}{\partial w_l})_i$
$(sparse\nabla w_l)_i$
$(\frac{\partial L}{\partial a_l})_i$
$(sparse\nabla a_l)_i$
Back-ReSprop (sparse conv)
Threshold
$(\frac{\partial L}{\partial y_{l+1}})_i$
∇f
$(\frac{\partial L}{\partial a_{l+1}})_i$
Backward pass

Figure 3. Training with ReSprop for layer l at iteration i.

gradients to zero (meProp Val Acc). MeProp has considerably less validation accuracy versus HG after 30 epochs of training. The gap between HG and meProp validation accuracy is more pronounced at higher sparsity percentages. Further, Table 2 shows the accuracy of ReSprop (using HG) improves further after 200 epochs of training CIFAR-10.

4. ReSprop: Reuse-Sparse-Backprop

This section describes ReSprop, an efficient back-propagation algorithm, which we developed to exploit the reusability of gradients. We reformulate the back-propagation convolutions based on the HG vector, which leads to sparse convolutions and a training speedup. The HG vector in Eq. 4 at iteration i can be split into two separated parts: One, ReHG ("Reused HG") the output gradient of the previous iteration $(\frac{\partial L}{\partial y_{l+1}})_{i-1}$ and is computed and stored before the current iteration; two, SpHG ("Sparse HG") the result of $Th[(\frac{\partial L}{\partial y_{l+1}})_i - (\frac{\partial L}{\partial y_{l+1}})_{i-1}]$. SpHG is sparse due to the threshold function. Using these definitions Eq. 4 can be rewritten as follows:

$$(HG_l)_i = (ReHG_l)_i + (SpHG_l)_i \tag{6}$$

By replacing the output gradient in Eq. 3 and 2 with the HG vector defined in Eq. 6 the back-propagation convolutions can be rewritten as:

$$(\frac{\partial L}{\partial w_l})_i = \underbrace{((ReHG_l)_i \otimes (a_l^t)_i)}_{①\ Pre\nabla w_l} + \underbrace{((SpHG_l)_i \otimes (a_l^t)_i)}_{②\ Sparse\nabla w_l} \tag{7}$$

$$(\frac{\partial L}{\partial a_l})_i = \underbrace{((ReHG_l)_i \otimes (w_l^t)_i)}_{①\ Pre\nabla a_l} + \underbrace{((SpHG_l)_i \otimes (w_l^t)_i)}_{②\ Sparse\nabla a_l} \tag{8}$$

Algorithm 1 ReSprop forward pass for l^{th} convolutional layer at iteration i.

```
1: for $l = 1$ to Layers do
2:     Receive: $random\ sample\ (\frac{\partial L}{\partial y_{l+1}})_{i-1}$
3:     $(y_{l+1})_i = (w_l)_i \otimes (a_l)_i$
4:     $(pre\nabla w_l)_i = (random\ (\frac{\partial L}{\partial y_{l+1}})_{i-1}) \otimes Avg(a_l^t)_i$
5:     $(pre\nabla a_l)_i = (w_l^t)_i \otimes (random(\frac{\partial L}{\partial y_{l+1}})_{i-1})$
6: end for
```

Algorithm 2 ReSprop backward pass for l^{th} convolutional layer at iteration i.

```
1: for $l$ = Layers to 1 do
2:     Receive:$(\frac{\partial L}{\partial y_l})_i$, $random\ sample\ (\frac{\partial L}{\partial y_{l+1}})_{i-1}$
3:     $Calculate\ (SpHG_l)_i$
4:     Receive: $(pre\nabla w_l)_i$ from forward pass
5:     $(\frac{\partial L}{\partial w_l})_i = (pre\nabla w_l)_i + (SpHG_{i,l} \otimes (a_l^t)_i)$
6:     Receive: $(pre\nabla a_l)_i$ from forward pass
7:     $(\frac{\partial L}{\partial a_l})_i = (pre\nabla a_l)_i + ((w_l^t)_i \otimes (SpHG_l)_i)$
8:     Update $(w_l)_i$ with $(\frac{\partial L}{\partial w_l})_i$
9:     Send $(\frac{\partial L}{\partial a_l})_i$ to previous layer
10: end for
```

Using $ReHG + SpHG$ in the back-propagation convolutions as shown in Eq. 7 and 8 allows us to break calculations into two parts labeled ① and ②. Part ① represents the precomputed portion and can be calculated in parallel with forward-propagation, before the current iteration's backward-propagation starts and part ② is where computation is saved using sparse convolution due to the sparsity of SpHG. We name the above algorithm ReSprop. We call the process for calculating part ① pre-ReSprop (Alg. 1) and the process for calculating part ② back-ReSprop (Alg. 2). Varying reuse percentage leads to different levels of sparsity in back-ReSprop. Thus, we name the sparsity generated by our algorithm reuse-sparsity (RS). As shown in Alg. 2 (lines 5 and 7), in ReSprop the back-propagation convolutions are sparse, and RS percentage is the main factor that defines the amount of computation reduction. In Section 5, we analyze the accuracy of ReSprop and show that at 90% RS, it loses negligible (less than 1.1%) accuracy for different datasets and has higher accuracy compared to DSG and meProp sparse training algorithms.

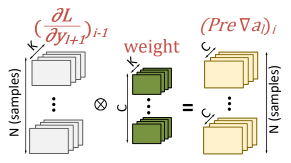
4.1. Stochastic Output Gradient

Storing the output gradients for an entire mini-batch at each iteration as implied by Eq. 4 to 8 creates a substantial memory overheads. We define **full mode** Resprop as a variant of ReSprop in which we store the output gradient for all samples in a minibatch. A simple approach for reducing the memory overheads and decreasing the computation in pre-ReSprop is to use the average output gradients of the previous mini-batch. We call this variant **average mode** ReSprop. In average mode, we add the extra step of computing average of gradients over the mini-batch. To avoid the extra step of averaging, stochastic sampling of the previous iteration's output gradient can be used in the ReSprop algorithm. We call this variant **stochastic mode** ReSprop. Our results indicate that using stochastic sampling of the output gradient does not decrease the accuracy of ReSprop compared to average or full mode. Table 2 shows the validation accuracy results for training ResNet-18 with CIFAR-10 using full, average, and stochastic mode variants of ReSprop after 200 epochs. Storing and using the output gradient vector of a random sample at each iteration significantly reduces the computation and memory cost of the ResProp. Below we use ReSprop as a shorthand for stochastic mode ReSprop.

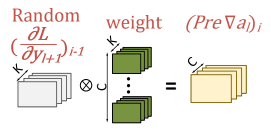
RS	Full (HG)	Avg	Stochastic
50%	94.54 ± 0.04	94.71 ± 0.06	94.69 ± 0.04
60%	94.38 ± 0.08	94.58 ± 0.03	94.66 ± 0.07
70%	94.36 ± 0.03	94.52 ± 0.04	94.53 ± 0.09
80%	93.18 ± 0.16	93.28 ± 0.12	93.51 ± 0.12
90%	91.10 ± 0.11	91.82 ± 0.07	91.43 ± 0.11
Baseline: 94.42 ± 0.08			

Table 2. Validation accuracy of full, average and stochastic ReSprop for ResNet-18 on the CIFAR-10 dataset for 200 epochs (batch size = 128, lr = 0.1, optimizer = SGD, avg of 3 runs).

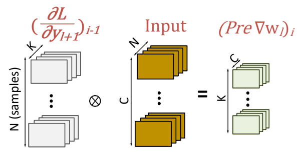
Algorithms 1 and 2 show the forward and backward pass calculations, respectively, for ReSprop. The convolutions needed for computing $pre\nabla a$ and $pre\nabla w$ in the full mode are shown respectively in Figure 4(a) and 4(c). We decrease the memory and computation overheads needed for convolutions in pre-ReSprop by a factor of mini-batch size when we use stochastic or average mode. The convolutions for stochastic mode is shown in Figure 4(b) and 4(d). For computing $pre\nabla a$ in Figure 4(b), one random output gradient ($K \times H \times W$) out of N samples is chosen and convolved with weights, producing one sample $pre\nabla a$, which then is replicated N times for all the N samples. Similarly, for computing $pre\nabla w$ in Figure 4(d), a random output gradient ($K \times H \times W$) out of N samples is chosen and reshaped into the desired shape ($K \times 1 \times H \times W$). Since in stochastic mode, we use the output gradient of a random sample, the output gradient is the same for all the convolutions for computing $pre\nabla w$. Thus, due to the distributive property of convolutions, we can average the inputs and then convolve the average input with a random gradient sample (Figure 4(d)). Figure 3 demonstrates the computation flow of ReSprop for the forward and backward pass. The Back-ReSprop box in the figure represents backward convolutions which are sparse (lines 5 and 7 in Alg. 2). The computation overhead of ReSprop for the forward pass computations (pre-ReSprop) is shown in Figure 6; this overhead is less than 2% for batch sizes larger than 128.



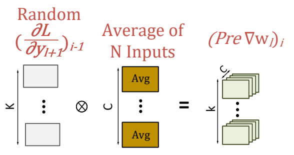
(a) Compute $pre\nabla a$ in full mode



(b) Compute $pre\nabla a$ in stochastic mode



(c) Compute $pre\nabla w$ in full mode



(d) Compute $pre\nabla w$ in stochastic mode

Figure 4. Back-propagation convolutions in stochastic mode compared to full mode for layer l at iteration i.

4.2. Warm Up

Narang *et al.* [45] and Zhu *et al.* [63] show that gradually increasing the sparsity percentage as training proceeds results in less drop in the model's final accuracy compared to maintaining a constant rate of sparsity during training. We apply the same approach and gradually increase the reuse-sparsity. We call this approach *warm up ReSprop* (W-ReSprop). In W-ResProp, we increase the sparsity percentage linearly in the first m ($m \ll number\ of\ epochs$) epochs until we get to the targeted reuse-sparsity. W-ReSprop helps the model adapt to gradient reuse, and it noticeably increases the network accuracy at high reuse-sparsities compared to base ReSprop. Results for W-ReSprop are shown and compared to base ReSprop in Section 5.

5. Evaluation

In this section, we present our experimental results of the ReSprop and W-ReSprop algorithms adapted to different datasets and architectures. Moreover, we quantify the theoretical computation reduction of ReSprop and simulate the speedup it achieves on a generic hardware accelerator designed to support sparse back-propagation.

5.1. Experimental Setup

We implement the ReSprop and W-ReSprop algorithms in PyTorch [47]. To evaluate our algorithms, we train three different widely used state-of-the-art architectures; ResNet-18, 34, 50 [19], Wide Residual Networks [61], and VGG-16 [53] on three different datasets: CIFAR-10, CIFAR-100

		CIFAR-100			CIFAR-10		
RS	**Algorithm**	**ResNet34**	**WRN-28-10**	**VGG-16**	**ResNet34**	**WRN-28-10**	**VGG-16**
50%	ReSprop	76.02 ± 0.15	81.45 ± 0.17	72.58 ± 0.23	95.85 ± 0.06	96.58 ± 0.09	93.35 ± 0.18
	W-ReSprop	76.4 ± 0.11	81.78 ± 0.16	72.79 ± 0.21	95.91 ± 0.05	96.93 ± 0.11	93.28 ± 0.19
60%	ReSprop	75.81 ± 0.15	80.44 ± 0.16	70.89 ± 0.22	95.25 ± 0.04	95.89 ± 0.11	93.18 ± 0.14
	W-ReSprop	76.01 ± 0.12	81.34 ± 0.15	72.45 ± 0.22	95.41 ± 0.09	96.79 ± 0.07	93.26 ± 0.15
70%	ReSprop	73.92 ± 0.18	78.34 ± 0.11	69.76 ± 0.19	95.01 ± 0.07	95.68 ± 0.08	92.63 ± 0.16
	W-ReSprop	75.60 ± 0.13	81.09 ± 0.15	71.98 ± 0.23	95.23 ± 0.09	96.13 ± 0.15	92.91 ± 0.17
80%	ReSprop	70.76 ± 0.15	76.87 ± 0.13	66.04 ± 0.29	94.17 ± 0.07	93.23 ± 0.08	91.90 ± 0.18
	W-ReSprop	75.44 ± 0.17	80.87 ± 0.14	71.88 ± 0.23	94.96 ± 0.13	95.93 ± 0.12	92.64 ± 0.17
90%	ReSprop	69.12 ± 0.13	75.06 ± 0.10	65.32 ± 0.21	91.61 ± 0.09	90.71 ± 0.15	90.01 ± 0.18
	W-ReSprop	**75.14** ± 0.16	**80.38** ± 0.17	**71.57** ± 0.24	**94.36** ± 0.07	**95.67** ± 0.11	**92.43** ± 0.18
Baseline		75.61 ± 0.16	81.29 ± 0.17	72.50 ± 0.21	95.13 ± 0.09	96.30 ± 0.11	93.25 ± 0.15

Table 3. Validation accuracy of ReSprop and W-ReSprop at different reuse-sparsity constraints on the CIFAR-10 and CIFAR-100.

[30] and ImageNet ILSVRC2012 [51]. For training, we use the SGD optimizer with momentum of 0.9, weight decay of 0.0001, initial learning rate of 0.1 and 5 to 8 warm up epochs for W-ReSprop. The baseline is trained with no sparsity or reusing. CIFAR-10 and CIFAR-100 datasets are trained for 200 epochs on a single GPU with a mini-batch size of 128. The learning rate is annealed by a factor of $(1/10)^{th}$ at the 80^{th} and 120^{th} epochs. We run each experiment with three different seeds and use the average value for all the results. The ImageNet dataset is trained for 90 epochs with a total mini-batch size of 256 samples on 4 GPUs (RTX 2080 Ti GPU). The learning rate is reduced by $(1/10)^{th}$ at the 30^{th} and 60^{th} epoch. The choice of hyper-parameters follows [19, 21]. For all evaluations in this section, we use the above setup, except in Section 5.3, where we study batch size impact and effect of the number of compute nodes on accuracy.

5.2. Accuracy Analysis

In this section, we provide a comprehensive analysis of the ReSprop and W-ReSprop algorithms and evaluate convergence and robustness on a wide range of models.

Accuracy on CIFAR10 and CIFAR100: Table 3 shows the accuracy of the ReSprop and W-ReSprop algorithms at different reuse-sparsity percentages on CIFAR-10 and CIFAR-100 datasets. We can see that ReSprop and W-Resprop algorithms achieve better accuracy than the baseline with reuse-sparsities of 50% and 60%, respectively. CIFAR-10, with fewer classification classes, is more robust to reuse gradients, and it suffers only a slight accuracy loss at 70% reuse-sparsity using the ReSprop algorithm. While the accuracy drop for reuse-sparsities higher than 70% is considerable in the ReSprop algorithm, it can be avoided by the addition of a warm up phase. For both CIFAR-10 and CIFAR-100, on three different architectures, W-ReSprop algorithm loses less than 0.95% validation accuracy at 90% and less than 0.7% at 80% reuse-sparsity. **Accuracy on ImageNet:** Table 4 shows the accuracy obtained by the ReSprop and W-ReSprop on ResNet34, VGG-16 and Wide-Resnet-50-2. The results indicate that unlike CIFAR datasets for which W-ReSprop and ReSprop algorithms outperform the baseline at reuse-sparsities lower than 70%, for the ImageNet dataset at 50% resue-sparsity ReSprop and W-ReSprop have less than 0.5% and 0.15% loss of accuracy, respectively. We observe that for the CIFAR-100 dataset, the W-ReSprop algorithm has better accuracy at high reuse-sparsities compared to the base ReSprop; the same trend holds for the Imagenet dataset. W-ReSprop at 90% reuse-sparsity has less than 1.1% accuracy loss in all three networks. For a fair comparison with W-ReSprop, we

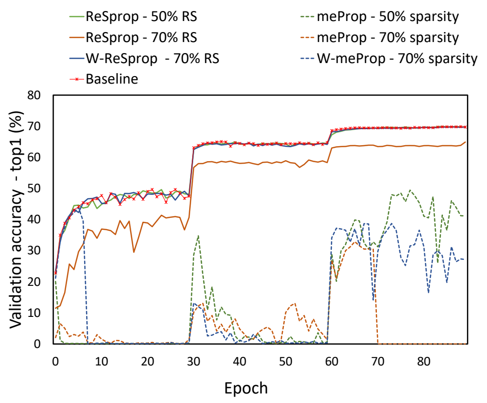


Figure 5. Top 1 validation accuracy of ReSprop, W-ReSprop, meProp and W-meProp algorithms for training ResNet-18 on the ImageNet dataset. The baseline is trained with no sparsity or reusing.

evaluate W-meProp, a variation of the meProp algorithm employing a warm up phase. Figure 5 demonstrates the validation curve of ReSprop, W-ReSprop, meProp and W-meProp algorithms on the ImageNet dataset for the Resnet-18 architecture. The validation curve indicates a significant

RS	Algorithm	ResNet34	WRN-50-2	VGG16
50%	ReSprop	73.08	78.69	70.09
	W-ReSprop	73.21	78.81	70.41
70%	ReSprop	67.12	73.34	68.73
	W-ReSprop	72.73	78.25	70.01
90%	ReSprop	63.78	67.72	60.76
	W-ReSprop	**72.44**	**77.93**	**69.46**
Baseline		73.34	78.88	70.50

Table 4. Top 1 validation accuracy of ReSprop and W-ReSprop algorithms at different reuse-sparsity constraints on the ImageNet dataset.

loss of accuracy for meProp and W-meProp. MeProp has validation accuracy of 32.56% at 50% sparsity while the ReSprop validation accuracy at 50% reuse-sparsity is 69.83% which is 0.03% less than the baseline. The W-Resprop algorithm at 50% reuse-sparsity gains 0.08% higher accuracy than the baseline and loses negligible accuracy of 0.7% at 70% reuse-sparsity.

5.3. Sensitivity Study

Deep and wide networks: Previous studies have shown that network depth and width affect network convergence [39, 52, 48]. Here, we study the effect of depth and width on the ReSprop algorithms. Table 5 shows the accuracy of ResNet-18, 34 and 50 for W-Resprop algorithm on CIFAR100 at 90% reuse percentage. We observe from the results that W-ReSprop converges to the state-of-the-art accuracy with minimal loss of accuracy for deep networks. At 90% reuse-sparsity W-ReSprop algorithm has an accuracy loss of 0.17% for ResNet-50. Similarly, the results for WRN-28-10 shown in Table 3, shows slight accuracy loss on training wide networks with the W-ReSprop algorithm.

	ResNet18	ResNet34	ResNet50
W-ReSprop 90%	73.26	75.15	76.67
Baseline	74.84	75.61	76.84

Table 5. Validation accuracy of ResNet-18, 34 and 50 on the CIFAR-100 dataset at 90% reuse-sparsity.

Impact of batch size: Here, we explore effects of batch size on the accuracy of ReSprop. Table 6 demonstrates that ReSprop converges with negligible accuracy loss for different batch sizes. ReSprop and W-ReSprop algorithms achieve higher accuracy for larger batch sizes. This behavior may be a result of including more samples increasing the likelihood similar features are present resulting in a higher correlation with the next iteration's gradients.

Distribute training across multiple compute nodes: Data parallelism is a popular way to accelerate training [31, 49]. To explore the impact of distributed training on accuracy, and still ignoring speedup, we evaluate ReSprop on

Batchsize	32	64	128
ReSprop 70%	72.94	73.48	73.92
W-ReSprop 90%	74.98	75.09	75.14
Baseline	76.12	75.88	75.61

Table 6. Validation accuracy of ResNet-34 on the CIFAR-100 dataset with different batch sizes of 32, 64 and 128.

multiple GPUs to compute gradient updates and then aggregating these locally computed updates. Below, we focus on training with multiple GPUs on a single machine by splitting the input across the specified GPUs. The ReSprop algorithm (Alg. 1 and 2) is applied during the training on each GPU independently. Table 7 shows the accuracy results for training ResNet-18 on ImageNet with a varied number of GPUs on a single machine. Since the ReSprop algorithm is applied to each GPU, the number of GPUs does not affect the model's accuracy trained with the ReSprop algorithm.

# GPUs in total	2	4	8
Batchsize in total	**128**	**256**	**512**
W-ReSprop 90%	68.73	68.81	68.61
Baseline	69.21	69.45	69.47

Table 7. Top 1 validation accuracy of ResNet-18 on the ImageNet dataset trained on 2, 4 and 8 nodes.

5.4. Speedup

In this section, we quantify the computation reduction, overheads, and the speedup of the ReSprop algorithm. Since we are using 5 to 8 epochs of whole training (90-200 epochs) for the warm up phase, the speedup for W-ReSprop would be the same order as the ReSprop algorithm.

Adaptive thresholding: The threshold operation can be implemented with $O(n)$ complexity. For each layer, if the reuse-sparsity of $(SpHG_l)_i$ becomes less than the targeted reuse-sparsity, we halve T_l (Eq (5)) to force more elements to zero and use the updated value of T_l for the next iteration. On the other hand, if the sparsity of $(SpHG_l)_i$ is more than the targeted reuse-sparsity, we increase T_l by doubling it. To accelerate the process of moving toward the desired T_l, we chose the initialization value of 10^{-7} for all the layers in all the experiments, based on the output gradient's distribution on the ResNet-18, 34 and 50 on CIFAR datasets. We experimentally find that for a given layer and a fixed reuse-sparsity, the threshold is almost constant during training. Thus, the threshold can be updated after a specific number of iterations, which reduces the computation overhead. The total computation overhead of adaptive thresholding, matrix additions, and subtractions in the ReSprop algorithm is less than 2.5% for both Imagenet and CIFAR datasets.

Pre-ReSprop overhead: As shown in Section 4, the ReSprop algorithm consists of pre-ReSprop and back-

ReSprop. Pre-ReSprop can be calculated in parallel with the original forward pass convolution. Figure 6 plots computation overhead (measured in terms of floating-point operations) added by ReSprop to the forward-pass at different batch sizes. This overhead is less than 2% for batch sizes larger than 128. We theoretically analyze the memory footprint by calculating ReSprop parameters that need to be stored and fetched. The results of the pre-ReSprop calculations and a random sample of the previous iteration's output gradient are stored and used in the back-ReSprop. We compute ReSprop memory footprint overhead by considering the adaptive threshold, pre-ReSprop, and back-ReSprop overheads. For the CIFAR and ImageNet datasets for batch sizes larger than 128, ReSprop has less than 16% memory footprint overhead compared to the total model parameters and the input activations' memory footprint for different architectures (ResNet18, 34, 50 and VGG-16).

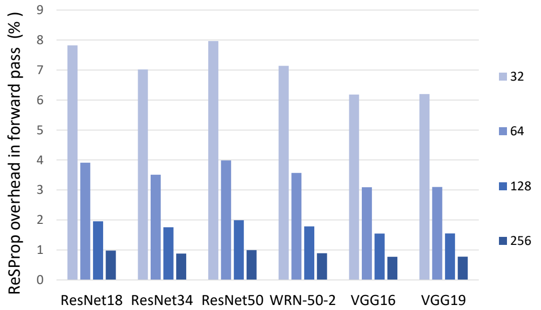


Figure 6. Computation overhead of ReSprop at forward pass (pre-ReSprop) for different batch sizes (ImageNet dataset).

Theoretical speedup: We evaluate the theoretical improvement in computational cost for forward and backward passes by comparing the number of floating-point operations with and without ReSprop. First row of Table 9 shows the theoretical speedup of ReSprop for the backward pass. Since ReSprop accelerates only the backward pass, the theoretical training (forward + backward) speedup can be calculated using Amdahl's Law [4]. Figure 7 shows the total training speedup considering the overheads of pre-ReSprop and thresholding. This analysis shows that at 90% reuse-sparsity, ImageNet can be trained $2.5\times$ to $3.0\times$ (on average $2.7\times$) faster using ReSprop. Among sparse training algorithms, DSG sparsifies back-propagation convolutions (Eq. 2 and 3). Table 8 shows the accuracy and speedup of DSG and W-ReSprop. W-ReSprop with the same sparsity percentage achieves higher accuracy and speedup. Reducing dimension for sparsifying gradients and inputs is the main reason for accuracy loss at high sparsities in DSG.

Accelerator for sparse back-propagation: We modify the SCNN [46] to support back-propagation convolutions and call the resulting architecture a generic sparse convolution accelerator (GSCN). We feed GSCN with sparse convolutions of ReSprop. To model performance of GSCN, we rely primarily on the DNNsim cycle-level simulator [14]. We extend this simulator to support GSCN. Table 9 shows the speedup we can gain on GSCN accelerator compared to GTX 1080 ti GPU by running standard training (second row) and ReSprop algorithm (third row) on GSCN.

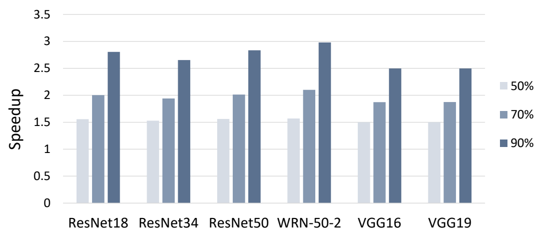


Figure 7. ReSprop training (forward+backward) speedup versus architecture for three reuse-sparsity percentages (ImageNet).

	ResNet-18		**WRN-8-2**	
Algorithm	**Speedup**	**Acc ↓**	**Speedup**	**Acc ↓**
DSG	2.2	3.88%	2.3	2.74%
W-ReSprop	2.7	0.51%	2.8	0.43%

Table 8. Validation accuracy and train speedup at 90% sparsity compared to dense training (CIFAR-10 dataset).

	ResNet18	**ResNet34**	**VGG-16**
Theoretical	9.83	9.68	9.34
GSCN+Baseline	1.32	1.81	1.27
GSCN+ReSprop	8.6	8.01	7.21

Table 9. Theoretical and GSCN speedup at backward pass computations with 90% resue-sparsity (ImageNet).

6. Conclusion

This work proposes Reuse-Sparsified Backpropagation for faster training by reusing the gradients during training. ReSprop sparsifies backward convolutions while adding minimal computation overhead to the forward pass. ReSprop and W-ReSprop can be used for training common network architectures and achieves average $2.7\times$ overall speedup in training with negligible loss in model accuracy when run on a generic convolution accelerator.

Acknowledgements We thank Mohammad Jafari for discussions and comments, and Aamir Raihan, Francois Demoullin, Deval Shah and Dave Evans for their feedback. This research funded in part by the Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) project financed under National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks grant number NETGP485577-15.

References

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394. ACM, 2017. 1

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. in 2016 ieee. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, volume 10, 2016. 1

[3] Zeyuan Allen-Zhu and Elad Hazan. Variance reduction for faster non-convex optimization. In *International Conference on Machine Learning*, pages 699–707, 2016. 2

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967. 2, 8

[5] Alexander G Anderson and Cory P Berg. The high-dimensional geometry of binary neural networks. *arXiv preprint arXiv:1705.07199*, 2017. 3

[6] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Advances in Neural Information Processing Systems*, pages 5145–5153, 2018. 1

[7] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*, 2017. 2

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 1

[9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014. 1

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012. 1

[11] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems*, pages 1646–1654, 2014. 2

[12] Aaron Defazio and Léon Bottou. On the ineffectiveness of variance reduced optimization for deep learning. *arXiv preprint arXiv:1812.04529*, 2018. 2

[13] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019. 1, 2

[14] Isak Edo, Omar Awad, Ali Hadi Zadeh, Dylan Malone Stuart, Alberto Delmas Lascorz, Milo Nikoli, and Andreas Moshovos. DNNsim: Deep Learning Accelerators Toolkit. `https://github.com/isakedo/DNNsim`. 8

[15] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014. 1

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016. 1

[17] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 2

[18] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems*, pages 164–171, 1993. 2

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 5, 6

[20] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision*, pages 1389–1397, 2017. 1, 2

[21] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017. 6

[22] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision*, pages 646–661. Springer, 2016. 1

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 1

[24] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013. 2

[25] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1(2):139–159, 2009. 3

[26] Ehud D Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990. 2

[27] Angelos Katharopoulos and François Fleuret. Biased importance sampling for deep neural network training. *arXiv preprint arXiv:1706.00043*, 2017. 1

[28] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942*, 2018. 1

[29] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014. 1

[30] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 6

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 1, 7

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 2

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

[34] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 2

[35] Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. Dynamic sparse graph for efficient deep learning. *arXiv preprint arXiv:1810.00859*, 2018. 1, 2

[36] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *IEEE International Conference on Computer Vision*, pages 2736–2744, 2017. 2

[37] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017. 1, 2

[38] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017. 1, 2

[39] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239, 2017. 7

[40] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *IEEE International Conference on Computer Vision*, pages 5058–5066, 2017. 1, 2

[41] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):2383, 2018. 2

[42] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. JMLR. org, 2017. 1, 2

[43] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016. 2

[44] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. *arXiv preprint arXiv:1902.05967*, 2019. 1, 2

[45] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017. 5

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2017. 1, 2, 8

[47] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 5

[48] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl Dickstein. On the expressive power of deep neural networks. In *International Conference on Machine Learning*, pages 2847–2854. JMLR. org, 2017. 7

[49] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *International Conference on Machine Learning*, pages 873–880. ACM, 2009. 7

[50] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International Conference on Machine Learning*, pages 314–323, 2016. 2

[51] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015. 6

[52] Or Sharir and Amnon Shashua. On the expressive power of overlapping operations of deep networks. *arXiv preprint arXiv:1703.02065*, 2017. 7

[53] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 5

[54] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454. ACM, 2017. 1

[55] Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3299–3308. JMLR. org, 2017. 1, 2, 3

[56] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015. 1

[57] Maria Vakalopoulou, Konstantinos Karantzalos, Nikos Komodakis, and Nikos Paragios. Building detection in very high resolution multispectral data with deep learning features. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 1873–1876. IEEE, 2015. 1

[58] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, pages 7675–7684, 2018. 1

[59] Bingzhen Wei, Xu Sun, Xuancheng Ren, and Jingjing Xu. Minimal effort back propagation for convolutional neural networks. *arXiv preprint arXiv:1709.05804*, 2017. 1, 2

[60] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016. 2

[61] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 5

[62] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017. 1

[63] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017. 5