ReSprop: Reused Sparsified Backpropagation

by

Negar Goli

B. Sc, Sharif University of Technology, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

June 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

ReSprop: Reused Sparsified Backpropagation

submitted by **Negar Goli** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Tor M. Aamodt, Electrical and Computer Engineering
Supervisor

Leonid Sigal, Computer Science
Supervisory Committee Member

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× resulting in a 2.7× 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× and 7.2× speedup in the backward pass on ResNet34 and VGG16 versus a GTX 1080 Ti GPU.

Lay Summary

Convolutional Neural Networks play an essential role in today's computer vision applications. However, to train these networks, one requires massive data and computational resources. Fortunately, more data is available due to the worldwide usage of the internet. But, the lack of efficient algorithms for training to reduce the computations hinders the progress. One way to decrease the computation in CNNs (Convolutional Neural Networks) is to produce sparsity in the computations. There are several methods, which sparsify the inference. However, due to the complexity of training, few works have studied sparsity in training for reducing the training computations.

In this thesis, we aim to solve the issue of excessive training time by developing a new training algorithm which reuses the gradients to sparsify computations. This method can achieve a substantial speedup on a specific hardware accelerator designed for sparse training.

Preface

This dissertation is based on a research project conducted by myself under the supervision and guidance of Professor Tor M. Aamodt. The work in this thesis was also presented in the paper ReSprop: Reuse Sparsified Backpropagation, accepted to appear in the oral presenation track at the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).

I assisted with defining the problem space and was responsible for deriving mathematical solutions, identifying challenges within this problem space, and designing and optimizing the algorithm to evaluate the proposed idea. Prof. Tor M. Aamodt provided valuable guidance and directions in identifying the research problems, developing solution methodologies, and documenting the results.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CIFAR	:	Canadian Institute for Advanced Research
Conv	:	Convolution
CNN	:	Convolutional Neural Networks
CUDA	:	Compute Unified Device Architecture
DNN	:	Deep Neural Networks
DRAM	:	Dynamic Random Access Memory
GPU	:	Graphics Processing Unit
GSCN	:	Generic Sparse Convolutional Neural Network
ReLU	:	Rectified Linear Unit
ReSprop	:	Reused Sparse Back-propagation
ResNet	:	Residual Networks
VGG	:	Visual Geometry Group

Acknowledgments

First and foremost, I am profoundly indebted to my parents in Iran for their unconditional love and blessings. Special thanks to my sister for her support and encouragement throughout my study.

Second, I would like to express my deep gratitude to Professor Tor M Aamodt, my research supervisor, for his patient guidance, enthusiastic encouragement, useful critiques of this research work, and generous financial support. I thank him for providing me with an excellent research atmosphere in his lab. The meetings and conversations were vital in inspiring me to think outside the box, from multiple perspectives to form a comprehensive study. I will remember and am deeply thankful for your wisdom and all the life lessons you taught me during my thesis.

Third, I am very fortunate and grateful to have excellent colleagues who offered me genuine and friendly support and assurance to carry out my research. I should particularly thank Md Aamir Raihan for several discussions and feedback on my research project. My special thanks to Francois Demoullin and Deval Shah for their constant encouragement and motivation, which helped me to get through one of the tough times of my life. I would also like to thank Dave Evans for his personal and professional support.

Finally, I would like to express my very great appreciation to Mohammad Jafari, UBC Ph.D. candidate, for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.

I also gratefully acknowledge the funding provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) that made my research

possible.

Dedication

To my father Mohammad-Reza Goli, mother Mahin Yaraghi, and sister Leily Goli.

With gratitude for your inspiration, love, and support.

Chapter 1

Introduction

We are witnessing an explosion in the use of Deep Neural Networks (DNNs), with major impact on the world's economic and social activity. At present, there is abundant evidence of DNN's effectiveness in areas such as classification, vision, and speech [34], [16], [68], [75], [64]. Of particular interest are Convolutional Neural Networks (CNNs), which achieve state-ofthe-art performance in many of these areas, such as image/temporal action recognition [46], [85] and scene generation [76]. However, state-of-the-art CNNs require extensive computational resources and a significant amount of time to be trained. The convolution operation is the primary computation source in CNNs.

Training a neural network consists of a forward pass and a backward pass. The backward pass has a higher computational cost compared to the forward pass. In the backward pass, the back-propagation algorithm is applied which is a common method for weights adjustment in conjunction with an optimization method such as gradient descent. The basic idea of the back-propagation algorithm is to propagate the error (the gradient in gradient descent) back along through the network and adjust the weights to correct the error. In brief, the backward pass shares similar computation patterns with the forward pass but involves approximately two-fold as many convolution operations for both error propagation and weight update, as shown in Figure 1.1. In this study, we try to reduce the computations in the backward pass. To have a better understanding of training, we need to also keep in mind that in practice, the most commonly used optimization method for CNN is stochas-

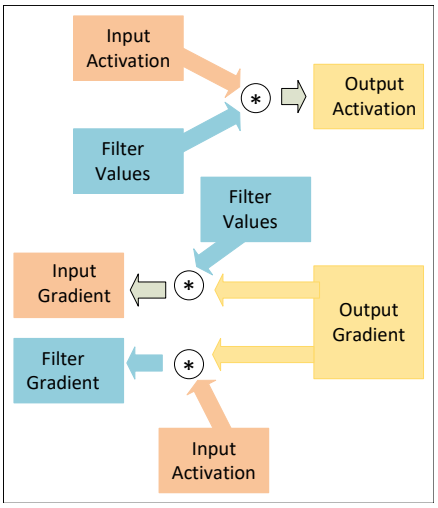


Figure 1.1: Forward and backward propagation through a convolutional layer during training.

tic gradient descent (SGD) [10], which randomly chooses a subset of the training data, called a minibatch, and updates parameters based on the average error of each iteration.

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 reducing internal covariate shift [37], parallelize training with data or model parallelism [17, 46], and importance sampling to reduce the variance of gradient estimates [44, 45]; (2) reducing the amount of computation per iteration using techniques such as stochastic depth to remove layers during training [35], randomized hashing to reduce the number of multiplications [87], quantization [8, 93, 98] and sparse training [21, 54, 90, 94]. We explore the second strategy and propose Reuse Sparse Backprop (ReSprop), a novel way to sparsify convolution computations during training[1].

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

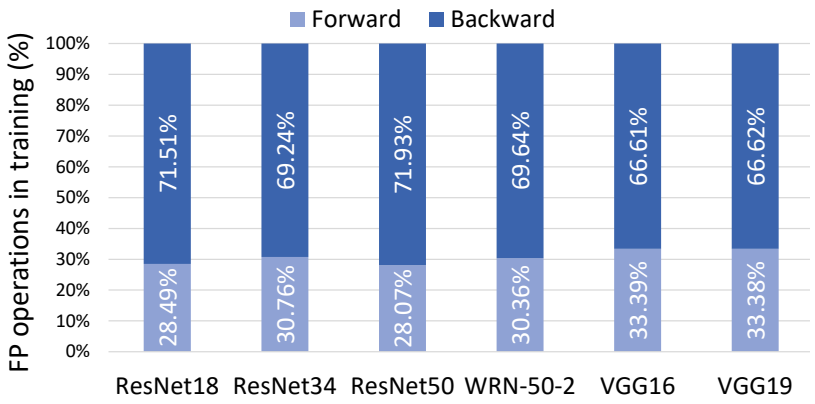


Figure 1.2: Percentage of floating point operations during the backward and forward pass in training different architectures.

1.1 Sparse Training

At their core, convolutions are parallel dot products and accumulations. Thus, sparse convolutions decrease computational cost by reducing the number of multiplication and addition operations. Recent related works [33, 58, 59, 61, 66] study different approaches to sparsifying inference, and many [1, 2, 13, 30, 72] propose accelerators to exploit sparsity in inference; however, there is limited work on sparse training [21, 54, 69, 90, 94].

In a forward-backward pass of a CNN during training, a convolutional layer requires three convolutions: one for forward propagation and two for backward propagation shown in Figure 1.1. Our measurements are shown in Figure 1.2 indicate that back-propagation consumes around 70% of the time during training.

MeProp [90, 94] and DSG [54] sparsify the backward pass convolutions using different sparsification methods. Since the sparsity produces by these methods is fine-grained, it might results in speedup on specialized hardware but not on an Nvidia GPU (before newly introduced Ampere GPU), which is the current preferred platform. Thus, the need for a specialized hardware accelerator for sparse inference and training is vital.

MeProp reduces the computational cost of training by sparsifying gradients in back-propagation calculations and DSG by graph selection method and dimension-

reduction search, compresses activations with elementwise unstructured sparsity, and accelerates vector-matrix multiplications (VMM). However, we observe that meProp fails to converge while training deeper networks or when using large datasets (Sections 4 and 7.1), and DSG loses more accuracy and achieves less training speedup compared to ReSprop.

ReSprop reduces the computation overhead of backward pass by reusing gradients to sparsify back-propagation convolutions. ReSprop overcomes prior limitations and loses less than 1.1% accuracy on large dataset such as ImageNet [82].

1.2 Gradient Reuse

In this section, we describe our main insight, which is reusing gradients, and the motivation behind it. We briefly explain prior works on reusing gradients and then move to our idea.

The stochastic variance reduction gradient (SVRG) method is proposed by Johnson and Zhang [40], and belongs to the class of stochastic methods using the so-called variance reduction technique [4, 19, 40, 62]. The common idea behind these methods is to use some full gradient of the past to approximate the future for general non-convex problems to reduce variance. Although the reusing proposed by SV methods is different from ours in many aspects such as the reusing strategy and the purpose of reusing, these methods motivate us to think of reusing gradients between iterations. We assume that at each iteration, not all the gradients will considerably change, and it would be possible that just a portion of them varies significantly, and the rest remains almost the same. We try to visualize the gradient's movement in a simple 3D example. As it is shown in this example, the gradient's movement in the first and second iterations are slightly different and the amount of change is almost the same.

Our observations (in Section 4.2) 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 5.1) exploits gradient reusability and sparsifies the gradients in the back-propagation convolutions up to 90% with less than 1.1% loss of accuracy on the ImageNet dataset. ReSprop has less than 2%

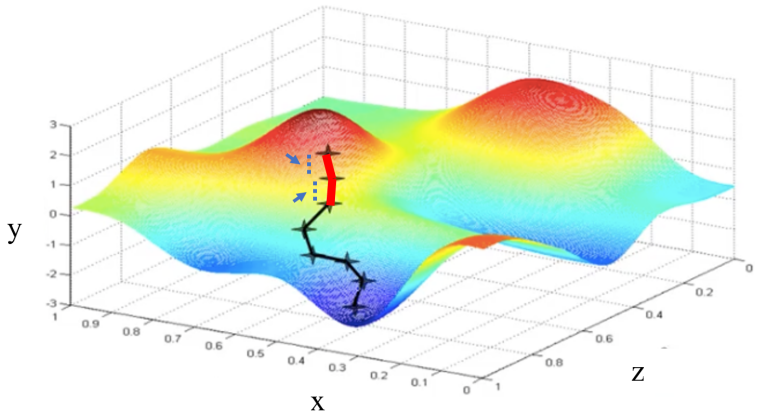


Figure 1.3: A simple example of gradient's movement in a 3D space. The gradient's movement in the first and second iterations are marked in red. The changes in the (y) axis is shown in blue dots.

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 [5], between 2.5× and 2.9× for the overall training process on different architectures.

1.3 GSCN Accelerator

In recent years accelerators, which are dedicated fixed-function peripherals designed to perform a single computationally intensive task over and over, have become an active area of research. Moreover, accelerators have been designed and adopted into programmable pipelines. Nvidia Tensor Core and ray tracing unit in Turing GPUs are some examples of it. Due to the high computation of convolutions, there are many algorithms for sparse convolutions and accelerators which try to exploits fine-grained sparsity produced by these algorithms. The current available GPUs are not able to exploit fine-grained sparsity, but the newly introduced NVIDIA Ampere GPU supports fine-grained sparsity of weights in the forward pass convolution.

The fine-grained sparsity introduces by ReSprop, can be accelerated by hardware similar to recently proposed inference accelerators [1, 2, 13, 30, 72]. Although there are many hardware accelerators for sparse inference, accelerators for sparse training have not been widely explored. Thus, we calculated the speedup achieved by ReSprop on our proposed custom hardware accelerator for sparse training.

Among available accelerator, Sparse CNN (SCNN) accelerator is an accelerator for inference, which exploits both weight and activation sparsity to improve both performance and power [72]. We propose a *Generic Sparse Convolutional Neural network (GSCN)* accelerator hardware architecture (Section 6) based on SCNN. GSCN is designed to accelerate sparse back-propagation convolutions. GSCN supports sparse back-propagation convolutions and overcomes the deficiencies of SCNN in cases of underutilization for small inputs and shortcomings caused by closely-sized input and filter. Our results (Section 7.5) show ReSprop on GSCN achieves 8.0× speedups versus a GPU on the backward pass of ResNet34.

1.4 Contributions

The contributions of this dissertation are:

- It shows that we can reuse about 90% of the gradients between consecutive iterations with minimal loss of accuracy.
- It introduces a new training algorithm called ReSprop, which by reusing the gradients, makes the backward pass computations sparse. For 90% sparsity, ReSprop accuracy loss is less than 1.1%. It achieves theoretical speedups between 9.3× and 9.8× for backward pass calculations and between 2.5× and 2.9× for the overall training process on different architectures.
- It shows that ReSprop on GSCN achieves between 7.2× and 8.6× speedups versus a GPU on the backward pass on different architectures.

1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 details the internal workings of a Convolutional Neural Network, and it is basic building blocks. It provides detailed background about training.
- Chapter 3 discusses related studies on pruning, sparse training, and reuse gradients.
- Chapter 4 describes our insight for reusing gradients and our reuse strategy and finally verifies it with experiments and comparison with meProp [90, 94].
- Chapter 5 proposes the ReSprop algorithm based on the reusing strategy and explain how reusing leads us to the sparse back-propagation calculations in ReSprop.
- Chapter 6 explains the GSCN architecture and dataflow.
- Chapter 7 demonstrates and analyzes accuracy results for different datasets, computation reduction, ReSprop overhead, and GSCN accelerator speedup.

Chapter 2

Background

To properly explain ReSprop and the experiments done to validate ReSprop, it is helpful to first briefly go over convolutional neural networks (CNNs), back-propagation's calculations in convolution layers and mini-batch training. This chapter gives an overview of convolutional neural networks (CNNs) and describes the commonly used CNNs topologies used in our experiments. It then explains training a network and briefly describes mini-batch training.

2.1 Convolutional Neural Network

The efficacy of Convolutional Neural Networks (CNNs) in image recognition is one of the main reasons why the world has woken up to the efficacy of deep learning. CNNs are powering major advances in computer vision, which has applications for self-driving cars, robotics, drones, security, medical diagnoses, and treatments for the visually impaired. CNNs are not limited to image recognition; however, they have been applied directly to text analytics and many other fields. A CNN architecture typically consists of convolution, activation, pooling, dropout, batch normalization, and fully-connected layers (Figure 2.1). In this section we explain the convolution layer, which is the main layer in CNNs in detail and briefly go over other layers.

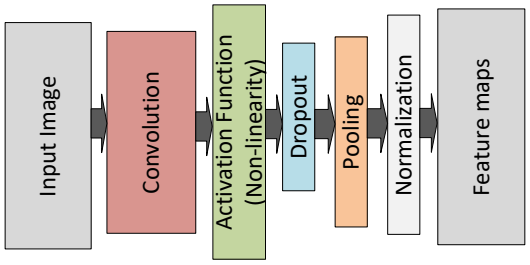


Figure 2.1: Different layers in CNN architecture.

2.1.1 Convolutional Layer

The predominant layers in CNNs are Convolutional layers. Each convolutional layer is composed of a 3-dimensional input with C channels ($width\ (W) \times height\ (H) \times channels\ (C)$) and K number of 3-dimensional ($width\ (R) \times height\ (S) \times channels\ (C)$) filters which in turn form a 3-dimensional $(W-R+1) \times (H-S+1) \times K$ output. These parameters are visualized in Figure 2.2. The width (W) and height (H) of an image are easily understood. The channels (C) are necessary because of how colors are encoded. Red-Green-Blue (RGB) encoding, for example, produces an image three layers deep. Each layer is called a channel, and through convolution, it produces a stack of output feature maps. The height and weight of the filters are smaller than those of the input volume. Convolution is a specialized kind of linear operation. Each filter is convolved with the input volume to compute an activation map made of neurons. In more details, the filter slides across the width and height of the input, and the dot products between the input and filter are computed at every spatial position. The output volume of the convolutional layer is obtained by stacking the activation maps of all filters along the depth dimension (channels) (Figure 2.2).

After a convolutional layer, the input is passed through a nonlinear transform such as tanh or rectified linear unit (ReLU), which will squash input values into a range between -1 and 1. One of the key challenges with images is that they are high-dimensional, which means they can consume a lot of time and computing power to process. Convolutional networks include stage designed to reduce the dimensionality of images. Filter stride is one way to reduce dimensionality; another

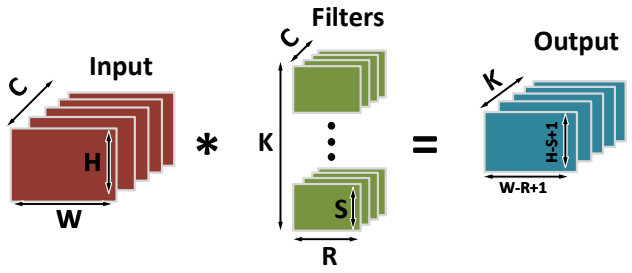


Figure 2.2: An input with C channels convolving with K filters to produce an output with K channels.

way is through downsampling, max-pooling, or subsampling. The pooling layer is almost part of any CNN network, and it mainly helps extract sharp and smooth features. It is also done to reduce variance and computations.

2.1.2 Activation Function

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding a bias to it. A network comprised of only linear activation functions is very easy to train, but cannot learn complex mapping functions. Linear activation functions are still used in the output layer for networks that predict a quantity (e.g. regression problems).

Increasingly, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions. Thus, the purpose of the activation function is to introduce non-linearity into the output of a neuron. While there are many activation functions, the most popular one is Rectified Linear Units (ReLU). Krizhevsky *et al* [46] have observed that deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units. Moreover, ReLU reduces likelihood of vanishing gradient. Generally, a neuron is turned on if the output of ReLU is greater than zero, and the neuron is turned off if the output of ReLU is zero.

2.1.3 Pooling

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.

A common approach to addressing this problem from signal processing is called downsampling. Thus, a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task. Downsampling can also be achieved with convolutional layers by changing the stride of the convolution across the image [88]. A more robust and common approach is to use a pooling layer. A pooling layer is added after a nonlinearity (e.g. ReLU). Pooling layer generally takes a patch of features and performs operations like average (Average Pooling) of all activation output values or maximum value for each patch of the feature map (max-pooling).

2.1.4 Dropout

Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or "dropped out". By dropping a unit out, the method temporarily removes it from the network, along with all its incoming and outgoing connections. Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation as a side-effect. As such, it is used as an alternative for regularizing the network [89].

2.1.5 Normalization

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. It has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks [37]. We explain mini-batch training later in Section 2.3.

By standardizing the activations of the prior layer, assumptions the subsequent layer makes about the spread and distribution of inputs during the weight update will not change, at least not dramatically. It has the effect of stabilizing and speeding up the training process of deep neural networks.

2.1.6 Fully-connected layer

Fully connected layers are an essential component of CNNs, which have been proven very successful in recognizing and classifying images. The CNN process begins with convolution and pooling, and the result of this process feeds into a fully connected neural network structure that drives the final classification decision. The fully connected part of the CNN goes through its own backpropagation process to determine the most accurate weights.

2.2 ResNet, WRN and VGG

Classic CNN network architectures were comprised simply of stacked convolutional layers. Modern architectures explore new and innovative ways of constructing convolutional layers in a way that allows for more efficient learning. Almost all of these architectures are based on a repeatable unit that is used throughout the network. In this study we use three common CNN networks called residual networks (ResNet) [32], wide residual networks (WRN) [96] and Visual Geometry Group (VGG) [86].

2.2.1 VGG

VGG is one of the most common deep convolutional network for object recognition. It was developed and trained by Oxford's renowned Visual Geometry Group (VGG), which achieved very good performance on the ImageNet dataset [86]. This network is characterized by its simplicity, using only 3×3 convolutional layers stacked on top of each other in increasing depth. Figure 2.3 shows the architecture for VGG-16.

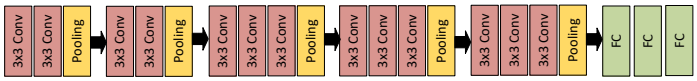


Figure 2.3: Architecure of VGG-16 network. Conv and FC denote convolution and fully-connected layers, respectively.

2.2.2 ResNet and WRN

A feedforward network with a single layer is sufficient to represent any function. However, the layer might be massive, and the network is prone to overfitting the data. Therefore, there is a common trend of using deep network architectures. However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem [73] as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitely small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly. ResNet solves this problem by introducing an identity shortcut connection that skips one or more layers (residual block) [32]. Typical ResNet models are implemented with double or triple layer skips that contain nonlinearities (ReLU) and batch normalization in between. Figure 2.4 shows an example of a residual block in the ResNet architecture.

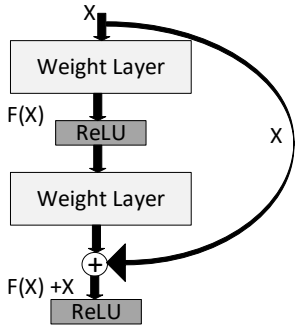


Figure 2.4: A simple residual block with skip connection for double layers.

However, deep residual networks are able to scale up to thousands of layers

and still improve the performance; each fraction of a percent of improved accuracy costs nearly doubling the number of layers. Thus, training very deep residual networks have a problem of diminishing feature reuse, which makes these networks very slow to train. To tackle these problems, Zagoruyko *et al* [96] proposes a novel architecture where they decrease the depth and increase the width of residual networks and called it wide residual networks (WRNs).

2.3 Training

In supervised training, which is our concern in this study, both the inputs and the outputs are provided. The network processes the inputs and compares its resulting outputs against the desired outputs (forward pass). Errors are then propagated back through the system, causing the system to adjust the weights which control the network (backward pass) [81]. This process occurs over and over as the weights are continually tweaked. Neural networks are trained using gradient descent, where the estimate of the error used to update the weights is calculated based on the training dataset. Therefore training can be separated into four distinct sections, the forward pass, the loss function, the backward pass, and the weight update. A loss function can be defined in many different ways, but a common one is MSE (mean squared error). Among different parts in training, the backward pass has the most computations and consumes about 70% of the time during training 1.2. In this study we focus on reducing the computations in the backward pass.

The goal of supervised training is to get to a point where the predicted label (output of the network) is the same as the training label. In order to get there, the amount of loss (L) needs to be minimized. In other words, the goal of training is to find out which weights most directly contributed to the loss (or error) of the network. This is the mathematical equivalent of a $\frac{\partial L}{\partial w}$ where w are the weights at a particular layer. We perform a backward pass through the network to determine which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. Once this derivative is computed, the weight gets updated accordingly. The learning rate (η) is a parameter that is chosen by the programmer. A high learning rate means that bigger steps are taken in the weight updates, and thus, it may take less time for the model to converge on an optimal set of weights

(2.1).

$$w_{i+1} = w_i - \eta \times \frac{\partial L}{\partial w_i} \tag{2.1}$$

2.3.1 Training Convolutions and Notation

In this section we explain the notation we use in the forward and backward pass convolutions. As we know in CNNs, convolution is the predominant operation in the forward and backward pass. The output of the l^{th} layer in the CNNs' forward-propagation is obtained by:

$$y_{l+1} = w_l \otimes a_l \tag{2.2}$$

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 [49]:

$$\frac{\partial L}{\partial a_l} = \frac{\partial L}{\partial y_{l+1}} \otimes w_l^t \tag{2.3}$$

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial y_{l+1}} \otimes a_l^t \tag{2.4}$$

2.3.2 Mini-batch Training

Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Therefore, mini-batch training applies the needed forward and backward pass calculations and updates the model parameters at each iteration (mini-batch) [11, 27, 50].

2.3.3 Momentum

A very popular technique that is used along with SGD is called Momentum [91]. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go. The equations of gradient descent are revised as follows.

$$\begin{aligned} z^{k+1} &= \beta z^k + \nabla f\left(w^k\right) \\ w^{k+1} &= w^k - \alpha z^{k+1} \end{aligned} \tag{2.5}$$

The first equations has two parts. The first term is the gradient that is retained from previous iterations. This retained gradient is multiplied by a value called "Coefficient of Momentum" (β) which is the percentage of the gradient retained every iteration. The change is innocent, and costs almost nothing. When $\beta = 0$, we recover gradient descent. But for $\beta = 0.99$, this appears to be the boost the network needs.

2.4 Sparse Convolution

As we described, the convolutional neural network (CNN) technique is built around the sharing of weights. It is influenced by the structural architecture of the human visual system. CNNs are based on ideas that utilize local connectivity between neurons and hierarchically organized transformation of the input. Nodes form groups of d-dimensional arrays known as feature maps. Each node in a given map receives inputs from a certain window area of the previous layer, which is referred to as its receptive field. The convolution operation results in a much sparser NN than the MLP.

Although CNN has fewer computations compared to MLP, it still has an extensive amount of computations. Thus, there are many different methods that sparsify the inputs and/or weights to make computations less on CNNs. In sparse CNN, due to the sparsity of the tensors, the multiplications in which one of the operands or both of them are zeros can be skipped. Many of the initial works on neural network focus on removing unimportant connections by sparsifying the weights (pruning).

2.4.1 Sparse Hardware Acceleretors

Recent works have examined how to support the processing of sparse weights and inputs in hardware efficiently. A variety of dedicated hardware accelerators for sparse matrix multiplication have been proposed. Most of the studies designed a parallel architecture comprising multiple processing elements using different dataflows and compression methods.

The DNN dataflow in the recent accelerators can be categorized based on the data handling characteristics:

1) Weight stationary: The weight stationary dataflow is designed to minimize the energy consumption of reading weights by maximizing the accesses of weights from the register file at the PE (processing element).

2) Input stationary: The input stationary dataflow is designed to minimize the energy consumption of reading inputs to PEs.

3) Row stationary: The row stationary dataflow assigns the processing of a 1-D row convolution into each PE for processing. It keeps the row of filter weights stationary inside the register file of the PE and then streams the input activations into the PE.

4) No local reuse (NLR): While small register files are efficient in terms of energy (pJ/bit), they are inefficient in terms of area. In order to maximize the storage capacity, and minimize the off-chip memory bandwidth, no local storage is allocated to the PE and instead all that area is allocated to the global buffer to increase its capacity.

SCNN accelerator, which is our focus in this study, supports the processing of convolutional layers in a compressed format. It uses a stationary input dataflow to deliver the compressed weights and activations to a multiplier array. We explain SCNN in detail in Section 6.3.

Chapter 3

Related Work

We categorize the related work into four different groups. First, we will discuss the pruning methods, which result in sparse networks. Second, we cover existing sparse training methods. Third, we talk about the reusing of gradients in prior training algorithms. Finally, we explain the hardware accelerators designed for sparse CNNs.

3.1 Dense to Sparse Networks by Weight Pruning

Creating sparse networks by eliminating the weights has an extensive history. LeCun *et al* [48]; Karnin [43]; Hassibi and Stork [31] present the early work of network pruning using second-order derivatives as the pruning criterion. Han *et al* [28] propose parameter magnitude as the pruning criterion and introduced the pipeline with three stages. These three stages are: 1) train a large, over-parameterized model (sometimes there are pretrained models available), 2) prune the trained large model according to a certain criterion, and 3) fine-tune the pruned model to regain the lost performance. The process of pruning and fine-tuning is often iterated several times, gradually reducing the network's size. Many papers propose slight variations of this algorithm. For example, some papers prune periodically during training [25] or even at initialization [52]. Others modify the network to explicitly include additional parameters that encourage sparsity and serve as a basis for the pruning criterion after training [67].

Pruning methods vary primarily in their choices regarding sparsity structure, pruning criterion, scheduling (when to prune), and fine-tuning.

3.1.1 Structure

Some methods prune individual parameters (unstructured pruning). Doing so produces a sparse neural network, which, although smaller in terms of parameter count, may not gain speedups using current available libraries and hardware. Other methods consider parameters in groups (structured pruning), removing entire neurons, filters, or channels to exploit hardware and software optimized for dense computation [33, 53].

3.1.2 Prunning Criterion

It is common to score parameters based on their absolute values, trained importance coefficients, or contributions to network activations or gradients. Some pruning methods compare scores locally, pruning a fraction of the parameters with the lowest scores within each structural subcomponent of the network (e.g., layers) [29]. Others consider scores globally, comparing scores to one another irrespective of the part of the network in which the parameter resides [24, 51]. In the same context, Structured Sparsity Learning (SSL) added group sparsity regularization to penalize unimportant parameters by removing some weights [95]. Li *et al* [53] proposed a one-shot channel pruning method using the L1 norm of weights for filter selection, provided that those channels with smaller weights always produce weaker activations. Recently, channel pruning alternatively used LASSO regression based channel selection and feature map reconstruction to prune filters [33]. Anwar *et al* [7]performes structured pruning in convolutional layers by considering strided sparsity of feature maps and kernels to avoid the need for custom hardware and uses particle filters to decide the importance of connections and paths. In contrast to previous pruning studies for deep deterministic models, Zahng *et al* [**?**]roposed a pruning approach for deep probabilistic models by using the mask of the weights.

3.1.3 Scheduling

Pruning methods differ in the amount of the network to prune at each step. Some methods prune all desired weights at once in a single step [57]. Others prune a fixed fraction of the network iteratively over several steps [29] or vary the rate of pruning according to a more complex function [25].

3.1.4 Fine-tuning

For methods that involve fine-tuning, it is most common to continue to train the network using the trained weights from before pruning. Alternative proposals include rewinding the network to an earlier state [24] and reinitializing the network entirely [57].

All the methods described in this section focus on gaining higher performance and/or accuracy at inference. These methods often involve a re-training phase, which, contrary to our motivation, increases training time.

3.2 Sparse Training

More recent studies try to find the sparse network during training through a prune, redistribute, and regrowth cycle. Mainly this group of studies prunes the network during training by adding more computation overhead to the training. The principal goal of most of them is to gain a higher accuracy using their pruned network for the inference. Bellec *et al* [9]; Mocanu *et al* [65]; Mostafa and Wang *et al* [69] propose different regrowth methods for sparsifying the networks through training. Dettmers *et al* [21] 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* [54] introduces 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* [90] and Wei *et al* [94] introduce meProp, an algorithm which targets computation reduction in training by sparsifying gradients. Meprop

computes an approximate gradient by keeping top-k values of the backward output gradient and masking the remaining values to 0. The forward propagation is computed as usual. However, during back-propagation, only a small subset of the full gradient is computed to update the model parameters. Thus, by using meProp algorithm, the backward pass convolutions explained in Section 2.3.1 can be redefined as follows:

$$\frac{\partial L}{\partial a_l} = Topk(\frac{\partial L}{\partial y_{l+1}}) \otimes w_l^t \tag{3.1}$$

$$\frac{\partial L}{\partial w_l} = Topk(\frac{\partial L}{\partial y_{l+1}}) \otimes a_l^t \tag{3.2}$$

Both of the above equations are sparse calculations due to the sparsity of output gradients. The amount of computation reduction and training speedup depends on the sparsity percentage of output gradients. The authors 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.

Golub *et al* [26] proposed Dropout which restricts randomly selected gradient updates during each training iteration and limits the set of weights that can be updated throughout the entire training process. This work decreases the number of weights that must be stored during the training process, but it does not make the training computations sparse.

3.3 Low precision neural network

Reducing data precision or quantization is another viable way to improve the computing efficiency of DNN accelerators. The recent TensorRT results show that the widely used NN models, including AlexNet, VGG, and ResNet, can be quantized to 8 bit without inference accuracy loss. However, it is difficult for such a unified quantization strategy to retain the network's accuracy when further lower precision is adopted. Many complex quantization schemes have been proposed, however, significantly increasing the hardware overhead of the quantization encoder/decoder and the workload scheduler in the accelerator design. There are many studies that tried to minimize the hardware overhead while reducing the computation and mem-

ory footprint. For example, one of the recent methods called Gist [38] reduces the memory footprint by encoding schemes of inputs through training, and it has just 4% performance overhead.

Our work focuses on accelerating training by sparsifying the training computation, and quantization and low periciosn methods are beyond the scope of our study.

3.4 Reusing Gradient

Full gradient ascent [12] with a constant step size achieves a linear convergence rate in the number T of iterations (i.e., parameter updates) [71]. However, each iteration requires N gradient computations, which can be too expensive for large values of N.

Stochastic Gradient (SG) ascent [11] overcomes this problem by sampling a single sample x_i per iteration, but a vanishing step size is required to control the variance introduced by sampling. Starting from SAG, a series of variations to SG have been proposed to achieve a better trade-off between convergence speed and cost per iteration, they called stochastic variance reduction (SVR) methods. SAG [80], SVRG [40], SAGA [19], Finito [20], and MISO [63] all are SVR methods proposed for smooth strongly-convex optimization problems. The common idea of these methods is to reuse past gradient computations to reduce the variance of the current estimate.

We breifly explain the most common SVR algorithms in more details. Both SAGA and SAG can be derived from a variance reduction viewpoint: here X is the SGD direction sample $f_j'\left(x^k\right)$, whereas Y is a past stored gradient $f_j'\left(\phi_j^k\right)$. In Eqs. 3.3, 3.4 and 3.5 you can see the differences between resuing methodologies. SAG is obtained by using $\alpha = 1/n$ (notation used in Eq. 3.3) whereas SAGA is the unbiased version with $\alpha = 1$ (Eq. 3.4). For the same ϕ 's, the variance of the SAG update is $1/n^2$ times the one of SAGA, but at the expense of having a non-zero bias.

$$\text{(SAG)} \quad x^{k+1} = x^k - \gamma \left[\frac{f_j'\left(x^k\right) - f_j'\left(\phi_j^k\right)}{n} + \frac{1}{n}\sum_{i=1}^{n} f_i'\left(\phi_i^k\right) \right] \tag{3.3}$$

[19].

$$\text{(SAGA)} \quad x^{k+1} = x^k - \gamma \left[f_j'\left(x^k\right) - f_j'\left(\phi_j^k\right) + \frac{1}{n}\sum_{i=1}^{n} f_i'\left(\phi_i^k\right) \right] \tag{3.4}$$

[19].

$$\text{(SVRG)} \quad x^{k+1} = x^k - \gamma \left[f_j'\left(x^k\right) - f_j'(\tilde{x}) + \frac{1}{n}\sum_{i=1}^{n} f_i'(\tilde{x}) \right] \tag{3.5}$$

[19].

The SVRG update (Eq. 3.5) is obtained by using $Y = f_j'(\tilde{x})$ with $\alpha = 1$. The vector $\tilde{x}$ is not updated every step, but rather the loop over k appears inside an outer loop, where $\tilde{x}$ is updated at the start of each outer iteration. Essentially SAGA is at the midpoint between SVRG and SAG; it updates the ϕ_j value each time index j is picked, whereas SVRG updates all of ϕ 's as a batch. The usage of SAG vs. SVRG is problem dependent. For example for linear predictors where gradients can be stored as a reduced vector of dimension $p-1$ for p classes, SAGA is preferred over SVRG both theoretically and in practice. For neural networks, where no theory is available for either method, the storage of gradients is generally more expensive than the additional backpropagations, but this is computer architecture dependent. Also having to tune one parameter instead of two is a practical advantage for SAGA.

Recent works explore the extension of SVR approaches to general non-convex problems [3, 79]. However, the faster theoretical convergence rate of the SVR methods is not a guarantee of better empirical performance in deep neural networks [18].

ReSprop reuses gradients in a different way than SVR. ReSprop reuses gra-

dients 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 $10\times$ computation reduction in back-propagation for different network architectures with varying widths and depths.

3.5 Hardware Accelerators for Deep Neural Networks

In the early stage of DNN accelerator design, accelerators were designed for the acceleration of approximate programs in general-purpose processing [23], or for small Neural Networks (NNs) [56]. Although the functionality and performance of on-chip accelerators were very limited, they revealed the basic idea of AI-specialized chips. Because of the limitations of general-purpose processing chips, it is of necessity to design specialized chips for AI/DNN applications.

On-chip accelerators:

The neural processing unit (NPU) [23] is designed to use hardwarelized on-chip NNs to accelerate a segment of a program instead of running all parts on a CPU. The hardware design of the NPU is quite simple. An NPU consists of eight processing engines (PEs). Each PE performs the computation of a neuron; that is, multiplication, accumulation, and sigmoid. Thus, what the NPU performs is the computation of a multiple layer perceptron (MLP) NN. The idea of using the hardwarelized MLP (NPU) to accelerate some program segments was very inspiring. If a program segment is frequently executed and approximable, and if the inputs and outputs are well defined, then that segment can be accelerated by the NPU. To execute a program on the NPU, programmers need to manually annotate a program segment that satisfies the above three conditions. Next, the compiler will compile the program segment into NPU instructions, and the computation tasks are off-loaded from the CPU to the NPU at runtime. Sobel edge detection and fast Fourier transform (FFT) are two examples of such program segments. The idea of the NPU was the inspiration for many of the later studies.

Stand-alone DNN/convolutional neural network accelerator:

For broadly used DNN and CNN applications, stand-alone domain-specific accelerators have achieved great success in both cloud and edge scenarios. Compared

with general-purpose CPUs and GPUs, these custom architectures offer better performance and higher energy efficiency. Custom architectures usually require a deep understanding of the target workloads. The dataflow (or data reuse pattern) is carefully analyzed and utilized in the design to reduce the off-chip memory access and improve the system efficiency. The DianNao series [14] and the tensor processing unit (TPU) [41], which are academic and industrial examples, respectively, are the two most popular stand-alone accelerators which we discuss more in details.

The DianNao series includes multiple accelerators with different features. DianNao is the first design of the series. It is composed of the following components: (1) A computational block neural functional unit (NFU), which performs computations; (2) An input buffer for input neurons (NBin); (3) An output buffer for output neurons (NBout); (4) A synapse buffer for synaptic weights (SB); (5) A control processor (CP). The NFU, which includes multipliers, adder trees, and nonlinear functional units, is designed as a pipeline. Rather than a normal cache, a scratchpad memory is used as on-chip storage because it can be controlled by the compiler and can easily explore the data locality. While efficient computing units are important for a DNN accelerator, inefficient memory transfers can also affect the system throughput and energy efficiency. The DianNao series introduces a special design to minimize memory transfer latency and enhance system efficiency.

On top of the stand-alone accelerators, a domain-specific instruction set architecture (ISA), called Cambricon [55], was proposed to support a broad range of NN applications. Cambricon is a load-store architecture that integrates scalar, vector, matrix, logical, data transfer, and control instructions. The ISA design considers data parallelism, customized vector/matrix instructions, and the use of scratchpad memory. The successors of the Cambricon series introduce support to sparse NNs.

Highlighted with a systolic array, Google published its first TPU paper (tpu1) in 2017. tpu1 focuses on inference tasks and has been deployed in Google's data center since 2015. The structure of the systolic array can be regarded as a specialized weight-stationary dataflow.

A DNN/CNN generally requires a large memory footprint. For large and complicated DNN/CNN models, it is unlikely that the whole model can be mapped onto the chip. Due to the limited off-chip bandwidth, it is of vital importance to increase on-chip data reuse and reduce the off-chip data transfer in order to improve com-

puting efficiency. During architecture design, dataflow analysis is performed, and special consideration needs to be taken. Eyeriss accelerator [15] explored different NN dataflows, including input-stationary, output-stationary, weight-stationary, and no-local-reuse dataflows, in the context of a spatial architecture and proposed the row-stationary (RS) dataflow to enhance data reuse.

The efficiency of DNN accelerators can also be improved by applying efficient NN structures. saprsifying the network, for example, makes the model small yet sparse, thus reducing the off-chip memory access. The NN quantization allows the model to operate in a low-precision mode, thus reducing the required storage capacity and computational cost.

3.5.1 Hardware Accelerator for Sparse Networks

As we discussed previous work shown that a large proportion of NN connections can be pruned to zero with or without minimum accuracy loss. Moreover, some of the activations are zero due to the Relu function. Thus, many corresponding computing architectures have also been proposed to exploit this sparsity. In particular, these works employ Run Length Encoding (RLE) to compress the inputs or the kernels. The RLE not only compresses the data but also allows to enhance the throughput by skipping the multiplications of the encoded data. While this approach seems appealing, combining the sparsity exploitation with other techniques (i.e. tiling, layer merging, etc. explained later in this section) makes efficient data flow management (and tiling) a non-trivial problem. Therefore, the existing architectures only exploit partial sparsity. Eyeriss [15], EIE [30], Cnvlutin [1], and Laconic [83] are some of the most well-known works that sought to remove multiplications by zero-valued activations and/or weights.

The authors of Cnvlutin achieved this by computing only non-zero inputs and using an "offset" buffer, alongside the input buffer, to store the indices of each input's corresponding weights after zero-skipping. A hardware controller fills the offset buffer on the fly such that it does not consume extra bandwidth. To further increase acceleration, Cnvlutin prunes near-zero outputs during inference to increase the sparsity of the next layer's input buffer. Eyeriss, EIE, and Laconic's authors achieved benefits from pruning using similar strategies to those employed by Cn-

vlutin's. EIE compresses and enhances the throughput only for the classification layers. It targets sparsity in both filters and feature maps only in fully-connected layers. Eyeriss exploits sparsity of all the layers, however, it cannot enhance the throughput (by skipping redundant computations), and it compresses only the inputs.

However, the special data format and extra encoder/decoder adopted in these designs introduce additional hardware costs. Some works discuss how to design NN models in a hardware-friendly way, such as by using block sparsity. Techniques that can handle irregular memory access and an unbalanced workload in sparse NN have also been proposed. For example, Cambricon-X [97] and Cambricon-S [99]address the memory access irregularity in sparse NNs through a cooperative software/hardware approach. Cambricon-X first marks non-zero neurons one by one, filters out zero-valued neurons, and then sends the neurons to the computational units for processing and eliminates unnecessary computation and weight storage. CNV [1] proposed a new data structure format for storing the inputs and outputs that enables the seamless elimination of most zero operand multiplications. The CNV storage format enables it to move the decisions on which computations to eliminate off the critical path. ReCom [39] proposes a ReRAM-based sparse NN accelerator based on structural weight/activation compression. Some other studies like Bit-pragmatic [2] skip zero-valued bits. Bit-pragmatic exploits activation sparsity in forward pass. In this work, they propose Pragmatic (PRA), a massively data-parallel architecture that eliminates most of the ineffectual computations on-the-fly. The idea behind it is using serial-parallel shift-and-add multiplication while skipping the zero bits of the serial input.

Thus, many previous works have studied accelerating the sparse convolution during forward propagation, and they focused on different data flows and compression methods during inference. However, none of them focus on an accelerator for sparse back-propagation convolutions.

The Sparse CNN (SCNN) accelerator [72] improves performance and energy efficiency by exploiting the zero-valued weights and activations during inference (forward pass). In order to evaluate ReSprop on a custom hardware accelerator, we propose an accelerator based on SCNN. Our accelerator (GSCN) can be used for both sparse forward and backward pass convolutions.

Chapter 4

Gradient Reuse

In this section, we explain our idea of reusing gradients and the insights behind it. First, we explain CNN convolutions and summarize the notation we will use throughout the remainder of this thesis. Then, we elaborate our hypothesis about gradients undergoing minor changes between consecutive iterations and propose a reuse strategy. Following this argument, we discuss our experiment designed to evaluate our resue strategy. Finally, we end this chapter with a more detailed comparison between our work and the most related prior work, meProp [90].

4.1 Preliminaries

In Section 2.3.1 we explained the convolutions calculated in training. In summary, the output of the l^{th} layer in the CNNs' forward-propagation (Eq. 4.1), and the back-propagation convolutions for calculating gradient of inputs (Eq. 4.2) and weights (Eq. 4.3) at the l^{th} layer is defined as follows (we use same notation as in Section 2.3.1):

$$y_{l+1} = w_l \otimes a_l \quad (4.1) \qquad \frac{\partial L}{\partial a_l} = \frac{\partial L}{\partial y_{l+1}} \otimes w_l^t \quad (4.2) \qquad \frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial y_{l+1}} \otimes a_l^t \quad (4.3)$$

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 the term "gradients" to refer to individual components of the gradient vector throughout this study.

4.2 Approach and Key Insight

Our approach to accelerate CNN training is to modify back-propagation convolutions. The *output gradient* vector and in turn the vectors dependant on it (Eq. 4.2 and 4.3) 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:

$$(HG_l)_i = (\frac{\partial L}{\partial y_{l+1}})_{i-1} + Th_l[(\frac{\partial L}{\partial y_{l+1}})_i - (\frac{\partial L}{\partial y_{l+1}})_{i-1}] \tag{4.4}$$

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:

$$\forall v_i \in V : \; u_i = \begin{cases} v_i & |v_i| > T_l \\ 0 & |v_i| \leq T_l \end{cases} \tag{4.5}$$

where u_i represents the elements of output vector $Th_l(V)$ and T_l is a per layer adaptive threshold. In Section 5.1, we explain how to use $(HG_l)_i$ to sparsify back propagation using ReSprop.

In Section 4.3, 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.

4.3 Angle Preservation and comparison with meProp

To study the correlation between HG and the original output gradient, we investigate the angle preservation property of the HG vector. We calculate the cosine similarity between HG and the original output gradient to measure the angle between these vectors. According to hyperdimensional computing theory [42], 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*[6] 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 4.1 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 ❶, 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

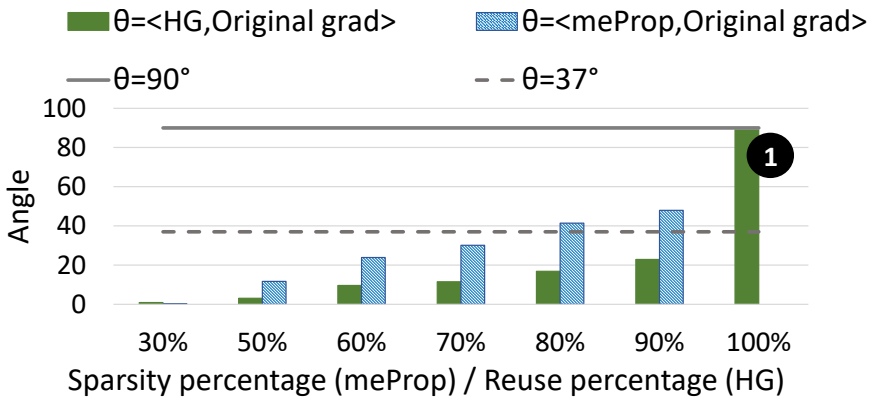


Figure 4.1: HG and meProp angles for different reuse percentages and sparsities, respectively. The angle is calculated by finding the average angle of all layers while training ResNet-18 on CIFAR-10 for 100 iterations (batch size=128).

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 4.1: 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).

output gradients and the HG vector to less than 37°. We compare this strategy with meProp [90] 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. 4.3 and 4.2 with the sparse output gradient. Figure 4.1 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 4.1

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 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 5.1 shows the accuracy of ReSprop (using HG) improves further after 200 epochs of training CIFAR-10.

Chapter 5

ReSprop Algorithm

5.1 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. 4.4 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. 4.4 can be rewritten as follows:

$$(HG_l)_i = (ReHG_l)_i + (SpHG_l)_i \tag{5.1}$$

By replacing the output gradient in Eq. 4.3 and 4.2 with the HG vector defined in Eq. 5.1 the back-propagation convolutions can be rewritten as:

$$(\frac{\partial L}{\partial w_l})_i = \underbrace{((ReHG_l)_i \otimes (a_l^t)_i)}_{① \; Pre\nabla w_l} + \underbrace{((SpHG_l)_i \otimes (a_l^t)_i)}_{② \; Sparse\nabla w_l} \tag{5.2}$$

$$(\frac{\partial L}{\partial a_l})_i = \underbrace{((ReHG_l)_i \otimes (w_l^t)_i)}_{① \; Pre\nabla a_l} + \underbrace{((SpHG_l)_i \otimes (w_l^t)_i)}_{② \; Sparse\nabla a_l} \tag{5.3}$$

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**

Using $ReHG + SpHG$ in the back-propagation convolutions as shown in Eq. 5.2 and 5.3 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 7.1, 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.

5.1.1 Stochastic Output Gradient

Storing the output gradients for an entire mini-batch at each iteration as implied by Eq. 4.4 to 5.3 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.

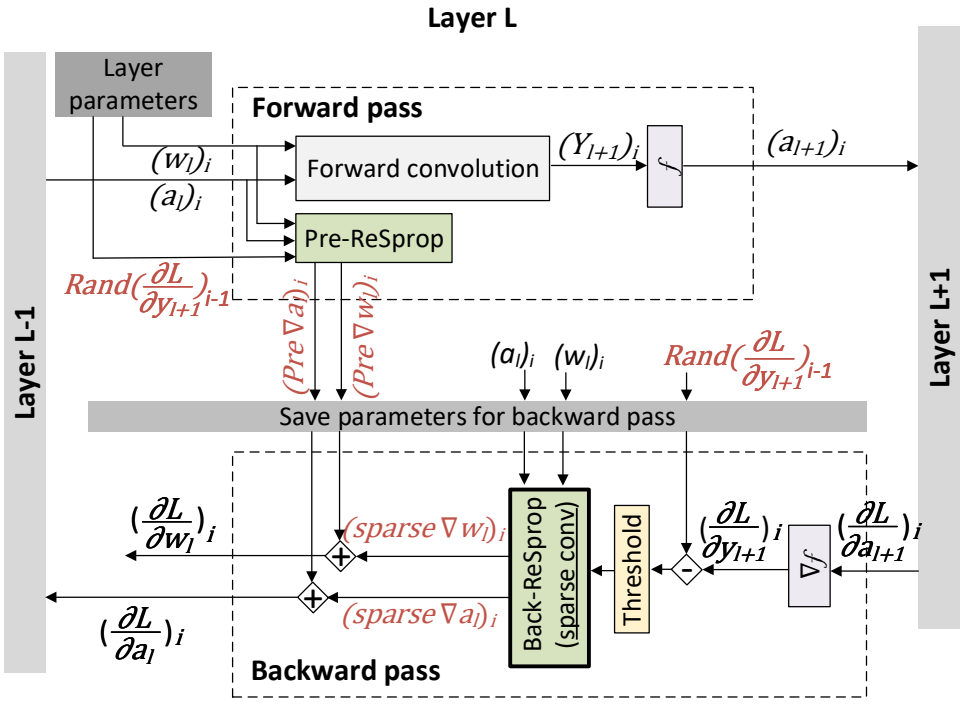


Figure 5.1: Training with ReSprop for layer l at iteration i.

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 5.1 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.

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 5.2a and 5.2c. We decrease the memory and computation overheads needed for convolutions in pre-ReSprop by

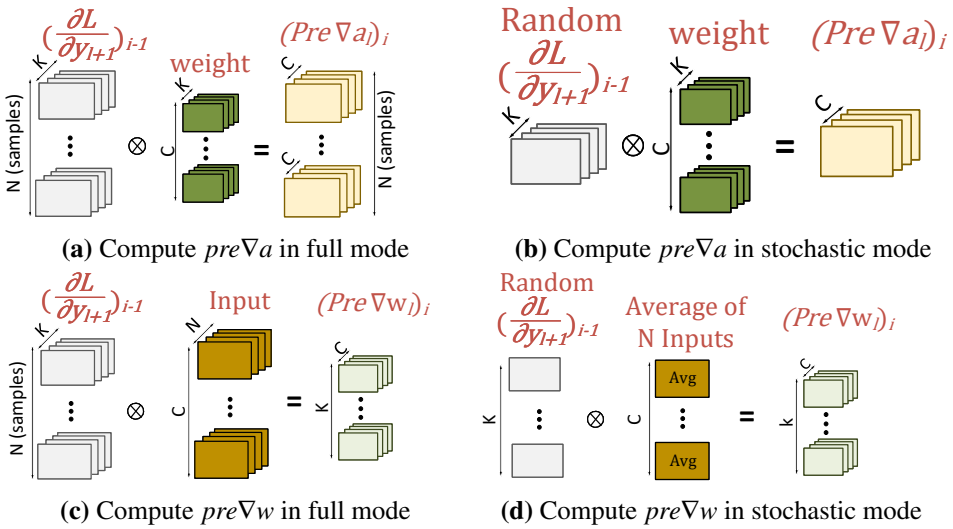
Algorithm 2 ReSprop backward pass for l^{th} convolutional layer at iteration i.

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

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 5.1: 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).

a factor of mini-batch size when we use stochastic or average mode. The convolutions for stochastic mode is shown in Figure 5.2b and 5.2d. For computing $pre\nabla a$ in Figure 5.2b, 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 5.2d, 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, instead of convolving a random sample with all N inputs, we



(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 5.2: Back-propagation convolutions in stochastic mode compared to full mode for layer l at iteration i.

can average the inputs and then convolve the average input with a random gradient sample as shown in Eq.5.4 for layer l and visulized in Figure 5.2d.

$$(pre\nabla w_l)_i = (random\ (\frac{\partial L}{\partial y_{l+1}})_{i-1}) \otimes Avg(a_l^t)_i \tag{5.4}$$

Figure 5.1 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 7.2; this overhead is less than 2% for batch sizes larger than 128.

5.1.2 Warm Up

Narang *et al* [70] and Zhu *et al* [100] 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 until reaching a targeted rate of 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 7.1.

Chapter 6

GSCN

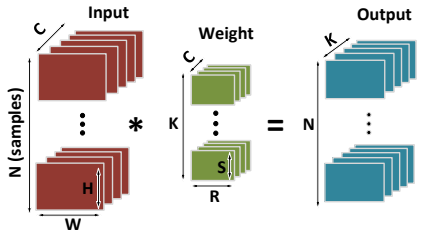
In recent years, custom hardware designs have been demonstrated to be an effective hardware platform to accelerate CNN inference and training. However, most existing architectures focus on sparse CNN inference. The architecture designed for sparse CNN inference is inefficient when executing sparse training. In this chapter, we explain our accelerator for sparse training, which is designed based on the SCNN accelerator.

6.1 Accelerator for Sparse Training

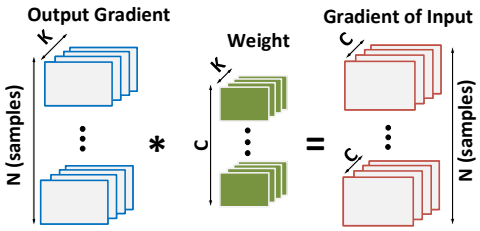
Sparse backward pass convolutions in ReSprop can be accelerated by exploiting the zero-valued elements in SpHG vector and zero-valued activations that arise from the common ReLU operator.

Among available accelerators, the SCNN accelerator is an accelerator for compressed-sparse convolutional neural networks proposed by [72]. It employs an efficient data flow that enables maintaining the sparsity in a compressed encoding. This architecture eliminates unnecessary data transfers and reduces storage requirements. Although SCNN improves the performance for sparse forward convolution, it suffers from underutilization when the input sizes are small. Moreover, SCNN can not be used for backward pass convolutions since while calculating the gradients of weights (Eq. 4.3), SCNN architecture introduces many unused products.

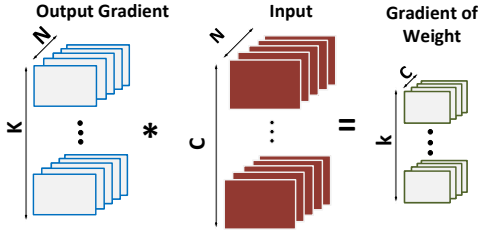
We propose a generic sparse convolution accelerator (GSCN) which improves



(a) Forward pass convolution



(b) Backward pass convolution for calculating gradients of inputs



(c) Backward pass convolution for calculating gradients of weights

Figure 6.1: Forward pass and backward pass convolutions for N input samples with C channels and K filters each with C channels.

SCNN by:

- Implementing different data and workload distribution among processing elements (PEs) to solve **PE underutilization**.
- Predicting **unused computations** produced by the SCNN architecture and skip them.

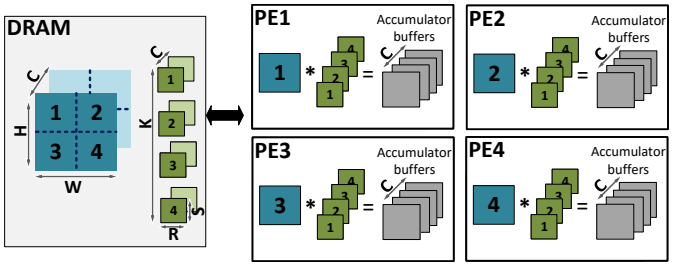
6.2 Computation of Convolutional Layers

As discussed in Section 2.1, the core operation in a CNN convolutional layer is a 2-dimensional sliding-window convolution of an $R \times S$ element filter over a $W \times H$ element input plane to produce a $(W - R + 1) \times (H - S + 1)$ element output plane. In CNN networks, filter sizes are usually small (3×3 or 1×1), and padding is applied to the input. Thus the output dimensions in many cases are close to the input size. There can be multiple (C) input planes, which are referred to as input channels, and multiple filters (K) can be applied to the same input to produce K output channels. In mini-batch training, a mini-batch of length N input planes is applied to the same filter volume. Figure 6.1a shows N inputs convolving with K filters as described in Section 2.1.

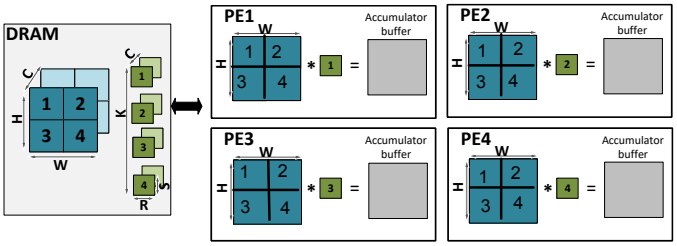
During back-propagation convolutions, the output gradient is convolved with weights, and input activations (Eq. 4.3 and 4.2). The output gradient has K channels. Thus, to convolve the output gradient with weights and input activations and have the desired shape, we need to reshape the inputs, weights, and gradients. As shown in Figure 6.1b the weights' dimensions are reshaped from $(K \times C \times R \times S)$ to $(C \times K \times R \times S)$. In this way, for each sample, K output gradients are convolved with K filters and produce one channel of input gradients. In Figure 6.1c the output gradients and inputs are reshaped to $(K \times N \times W' \times H')$ and $(C \times N \times W \times H)$, respectively. Thus, N output gradients are convolved with N inputs to produce one channel of weight gradients. These reshapings can be done before saving the weights, gradients, and inputs into the GSCN buffers.

6.3 Background on SCNN

Since GSCN is based on SCNN, in this section, we discuss the SCNN data flow and architecture in detail. The baseline data flow is based on SCNN's data flow. The SCNN data flow employs an input stationary computation order in which an input activation is held stationary at the computation units as it is multiplied by all the filters needed to make all its contributions to each of the K output channels. SCNN also factors the K output channels into K/Kc output-channel groups of size Kc, and only store weights and outputs for a single output-channel group at a time



(a) Data and workload distribution over four PEs for first channel in SCNN.



(b) Data and workload distribution over four PEs for first channel in GSCN.

Figure 6.2: Figure shows an example of SCNN and GSCN data and workload distribution while having four PEs. Input and filter assignment to each PE for SCNN and GSCN are different.

inside the weight and accumulator buffers. In this section, we try to explain SCNN dataflow and architecture by referring to Figure 6.3 for SCNN architecture, 6.2a for data and workload distribution and Alg. 3 for SCNN dataflow and operations.

SCNN dataflow requires buffers for filters and inputs, and an accumulator buffer to store the partial sums of the output activations. These buffers are shown in Figure 6.3 ①. SCNN employs a tiling strategy to spread the work across an array of PEs (P is the number of available PEs) so that each PE can operate independently. The $W \times H$ input plane breaks into smaller $\frac{W \times H}{P}$ element tiles ($Wt \times Ht$) that are distributed across the PEs. Thus each PE would get $Wt \times Ht$ portion of the input. All the Kc weights are broadcast to the PEs, and each PE operates on its own subset of the input and output space (Alg.3 line 2). The PEs need to communicate with each other since they contain incomplete, partial sums at the edge of tiles (halos). As shown in ④ Figure 6.3, there is a specific unit for Relu and compression operations and this unit also takes care of finding the haloes and communications

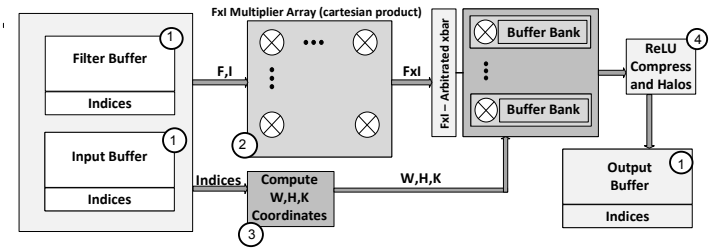


Figure 6.3: SCNN PE microarchitecture employing SSCN data flow [72].

between PEs. In SCNN from the filter buffer, a vector of F filter-weights is fetched, and with a vector of I inputs fetched from the input activation buffer are delivered to an array of $F \times I$ multipliers to compute a full Cartesian product (CP) of output partial-sums; as shown in Figure 6.3 ② and Alg.3 line 11. This all-to-all operation is useful since each fetched weight is reused (via wire-based multicast) over all I activations; each activation is reused over all F weights. The dataflow for each PE is shown in Alg. 3. As it is shown in lines 2 and 7, each PE will get a portion of whole input $Wt \times Ht$ and all Kc filters. Figure 6.2a further demonstrates data and workload distribution between PEs. In this figure, we assume an SCNN architecture with a total of 4 PEs ($p = 4$). Thus, each PE will get 1/4 of the input channel.

As explained SCNN uses the cartesian product to exploit the parallelism of many multipliers within a PE, and it fetches a vector of F filter-weights from the weight buffer, and a vector of I inputs from the input activation buffer. These values will multiply in an array of $F \times I$ multipliers to compute a full cartesian product of output partial-sums. SCNN will exploit the same property to make computation efficient on compressed-sparse weights and input (Figure 6.3). Parallel to the non-zero cartesian product, the coordinates in dense output is computed using the indices from the sparse-compressed filters and inputs as shown in Figure 6.3 ③ (Alg. 3 lines 12, 13 and 14). The $F \times I$ products are delivered to an array of A accumulator banks, indexed by the output coordinates (Alg. 3 line 15). To reduce the contention among products hashed to the accumulator bank, A is set to be larger than $F \times I$. SCNN shows that $A = 2 \times F \times I$ sufficiently reduces accumulator bank

Algorithm 3 SCNN dataflow for each PE

BUFFER W-buf$[C][Kc*R*S/F][F]$
BUFFER In-buf$[C][Wt*Ht/I][I]$
BUFFER Acc-buf$[Kc][Wt+R-1][Ht+S-1]$
BUFFER Out-buf$[K/Kc][Kc*Wt*Ht]$
for $k' = 0$ to $K/Kc-1$ **do**
  for $c = 0$ to $C-1$ **do**
    for $a = 0$ to $(Wt*Ht/I)-1$ **do**
      In$[0:I-1]$ = In-buf$[c][a][0:I-1]$;
      for $w = 0$ to $(Kc*R*S/F)-1$ **do**
        $wt[0:F-1]$ = W-buf$[c][w][0:F-1]$;
        for (parallel for)$(i = 0$ to $I-1) \times (f = 0$ to $F-1)$ **do**
          $k = Kcoord(w,f)$;
          $x = Xcoord(a,i,w,f)$;
          $y = Ycoord(a,i,w,f)$;
          acc-buf$[k][x][y] += in[i]*wt[f]$
        end for
      end for
    end for
  end for
  Out-buf$[k'][0:Kc*Wt*Ht-1]$ = Acc-buf$[0:Kc-1][0:Wt-1][0:Ht-1]$
end for

contention. The output then goes through ReLU and compression, and the partial sums at the tile edges are computed and sent to other PEs (Figure 6.3 ④).

6.3.1 SCNN Limitations

In this section, we describe the SCNN limitations. As we described, SCNN dataflow and architecture are computation efficient for sparse inference. However, SCNN has two main problems:

1) SCNN data flow performs well when the input dimension ($W \times H$) is big enough to utilize all the PEs. For most of the classification networks, the size of the input and accordingly, the size of the output and gradient of output in deeper layers become small. Thus, tiling (dividing) the small input size across PEs ($Wt \times$

Ht) in SCNN data flow will cause significant underutilization of PEs and loss of performance for most of the state of the art classification networks in deep layers. For example, the 16x16 input can not be divided between 64 PEs, and many PEs would be idle.

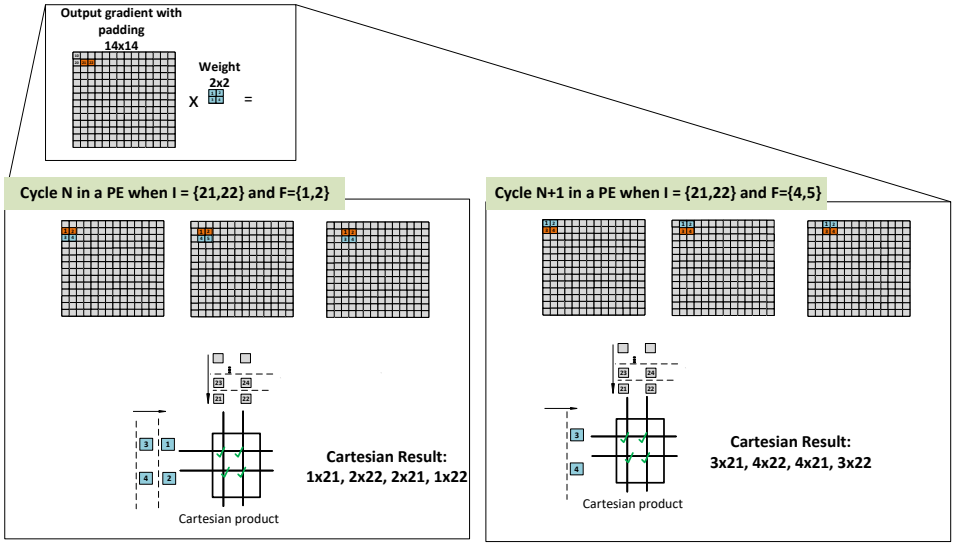


Figure 6.4: When the size of filter is smaller than input size, SCNN architecture has negligible amount of unused products.

2) The SCNN cartesian product works well for the forward pass convolutions since the filter dimensions ($R \times S$) are smaller than input dimensions ($W \times H$). However, each output product in the cartesian product mostly yields a useful partial sum. However, in gradients of weights convolution (Figure 6.1c) the input window is almost the same size as the output gradient. Thus many of the output products from the cartesian product would not be used, so we call them unused products. In Figure 6.5, we demonstrate the unused products produces when the size of input (14×14) and gradients (12×12) are close. In this example, the size of F and I is 2. Thus, at each cycle, the cartesian product produces four products. As shown in the figure after the first cartesian product (cycle N), there is no need to slide the input window further, and we need to skip the rest of the computations (cycle N+1) and other upcoming cycles till we read the next filter row into the F or read the next I

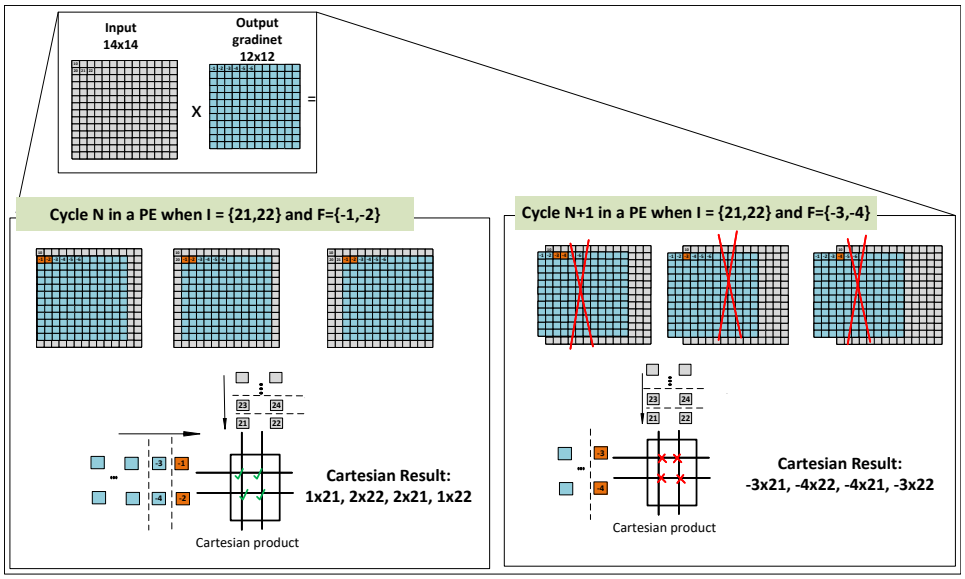


Figure 6.5: When the size of two operands are close in the convolution the SCNN architecture produces many unused products.

input activations.

The number of unused products is minor when size of the filter is smaller than the input size (Figure 6.4). The forward pass convolutions and one of the convolutions in the backward pass, which calculated gradient's of inputs, are in this category. The cartesian product for two cycles convolving gradients (14×14) with filters (2×2) in SCNN is shown in Figure 6.4 for these cases. Size of F and I are two and four parallel products computed in each cartesian product. As shown in this example, unlike Figure 6.5, the cartesian products are useful in both cycles and are part of the convolution calculations.

Table 6.1 shows the percentage of the cartesian product (4×4) that its products are not useful (all the 16 products are not used) over the total number of cartesian products in for gradient of weight convolution using SCNN architecture. I and F are four, and SCNN has in total 64 PEs.

	ResNet18	ResNet34
Unused cartesian products (%)	64.91%	69.6%

Table 6.1: The table shows percentage of unused cartesian product over total cartesian product in SCNN architecture for gradient of weights convolution.

6.4 GSCN Data Flow

We propose a Generic Sparse Convolutional Neural network (GSCN) with slight differences from SCNN data flow to solve the first problem mentioned in Section 6.3.1. This data flow employs the input stationary the same as SCNN, but it overcomes the underutilization problem. In GSCN's data flow, the input is not divided, but instead, the whole input is broadcasted to all the PEs. In this way, overcome the underutilization problem. In GSCN, instead, a subset of filters is assigned to each PE as follows. Thus, the size of input and filter buffer is different form of SCNN. The input buffer is $P\times$ larger and the filter buffer is $P\times$ smaller.

Assume the number of PEs is P, and we have K filters which same as SCNN are grouped to Kc filters. In GSCN, each PE gets $Ks = *\frac{Kc}{p}$ filters, and the partial sums are stored in the accumulator buffers. In GCSN's data flow, there is no need to communicate between PEs since each PE contains complete partial sums. Alg. 4 shows the GSCN dataflow. As it is shown in lines 1, 2, and 4, the size of buffers is different from SCNN. At line 5, GSCN's loop is over Ks filters instead of Kc filters, and at line 7, the loop is going over all the input elements. Note that the rest of the algorithm is almost the same as SCNN apart from the accumulator buffer size (lines 15 and 20).

Figure 6.2b further demonstrates our proposed data flow. In this example, it is assumed that GSCN architecture has 4 PEs. As it is shown, each PE gets whole input ($W \times H$), while filters are divided between PEs.

GSCN, like SCNN, uses available compression methods [30, 92] to compress sparse filters and inputs to reduce both arithmetic operations and data movement. The GSCN encoding includes a data vector consisting of the non-zero values and an index vector that includes the number of non-zero values followed by the number of

Algorithm 4 GSCN dataflow for each PE

1: BUFFER W-buf$[C][Ks*R*S/F][F]$
2: BUFFER In-buf$[C][W*H/I][I]$
3: BUFFER Acc-buf$[Ks][W+R-1][Ht+S-1]$
4: BUFFER Out-buf$[K/Ks][Ks*W*H]$
5: **for** $k' = 0$ to $K/Kc-1$ **do**
6: **for** $c = 0$ to $C-1$ **do**
7: **for** $a = 0$ to $(W*H/I)-1$ **do**
8: In$[0:I-1]$ = In-buf$[c][a][0:I-1]$;
9: **for** $w = 0$ to $(Ks*R*S/F)-1$ **do**
10: $wt[0:F-1]$ = W-buf$[c][w][0:F-1]$;
11: **for** (parallel for)$(i = 0$ to $I-1) \times (f = 0$ to $F-1)$ **do**
12: $k = Kcoord(w,f)$;
13: $x = Xcoord(a,i,w,f)$;
14: $y = Ycoord(a,i,w,f)$;
15: acc-buf$[k][x][y] += in[i]*wt[f]$
16: **end for**
17: **end for**
18: **end for**
19: **end for**
20: Out-buf$[k'][0:Ks*W*H-1]$ = Acc-buf$[0:Ks-1][0:W-1][0:H-1]$
21: **end for**

zeros before each value. The 3-dimensional $R \times S \times K$ filter is effectively linearized, enabling full compression across the dimension transitions. The activations are encoded in a similar fashion but across the $H \times W \times C$ dimensions. As the fitters are divided among the PEs, each tile of compressed filters is actually $R \times S \times Ks$.

The total size of the accumulator buffer for GSCN ($*\frac{Kc}{p} \times output's\ size$) does not differ from SCNN accumulator size ($Kc \times \frac{output's\ size}{p}$). Specifically, the input buffer size needs to be P times larger than SCNN, while the weight buffer size is P times smaller than the SCNN weight buffer. This is also visible by comparing the first four lines of Alg. 3 and Alg. 4.

To avoid having a significant input buffer size, GSCN can divide the input space into sub-volumes so that the collection of PEs operates on a sub-volume of

the inputs at a time. In this way, DRAM accesses for one temporal portion can be hidden by pipelining them in tandem with the computation of another portion. The sub-volume input space implementation is part of our future works.

6.5 GSCN Architecture

In this section we discuss the GSCN's architecure which is based upon SCNN but tackles SCNN second limitation (unused products) and improves the performance.

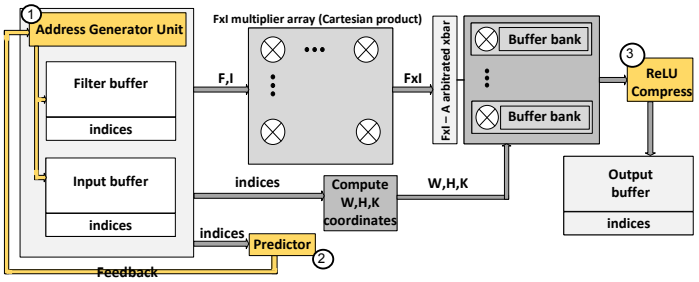


Figure 6.6: GSCN PE microarchitecture employing GSCN data flow. The GSCN PE microarchitecture is built upon SCNN, the units added or changes in GSCN have been shown with yellow color.

To prevent unused calculation, GSCN proposes a predictor (Figure 6.6 ②) that predicts unused products after loading inputs and filters' indices into the coordinate computer. This predictor sends feedback to the input and filter buffer's address generation unit ①. In SCNN and GSCN as well, the inputs are stored in $(W \rightarrow H)$ and filters in $(R \rightarrow S \rightarrow K)$ sequence. In Alg. 5 the filter storage sequence is shown. The first outer loop is over width, and the last loop is over filters. Thus, the width is always increasing while reading weights from filter buffer.

GSCN fetches a vector of I inputs from the input buffer, performs the cartesian product for all the filters in the filter buffer ($F \times I$ each time), and then moves to the next I inputs. Note that input and filter are operands for forward pass convolutions, and in the back-propagation convolutions, we have gradient and input's of the previous layer, and gradients and weight.

Algorithm 5 GSCN storage order for filter elements in the buffer for each channel

1: $Ks = \frac{Kc}{Number\ of\ PEs}$
2: **for** $r = 1$ **to** R **do**
3: **for** $s = 1$ **to** S **do**
4: **for** $k = 1$ **to** Ks **do**
5: FilterBuffer.pushback(filter(k, s, r))
6: **end for**
7: **end for**
8: **end for**

As explained while loading the filters from the filter buffer, the filter's width will never decrease since the outer loop is over width while storing the filters in the filter buffer (Alg. 5 line 1). The coordinate computer computes the output coordinates for a given $F \times I$. Assume the input indices are w and h, and weight indices are s and r. If the calculated output coordinate becomes negative ($(w-r) < 0$ or $(h-s) < 0$), it means that a particular output product is not going to be used in the convolution as shown in figure 6.5.

Thus, when the width of all output coordinates $(w-r)$ produced by a cartesian product (for a given $F \times I$) are negative, it indicates that all next F filter vectors are going to produce unused products with that particular I vector.

If none of the current F filters vectors are useful, the predictor skips all the upcoming products until we move to the next I input vector. Feedback signal sent to the address generator unit changes the address pointer so that the next I input vector and the first F filter from filter buffer be fetched.

This method will help us to skip many of the unused cartesian products and increase performance. The PE architecture is pipelined, and the predictor is placed right after loading the inputs; thus, our prediction will be acted upon with one cycle delay. Figure 6.6 demonstrates the microarchitecture of a GSCN PE, including a filter buffer, input/output buffer, address generation unit, a multiplier array, a coordinate computation, a predictor, a scatter crossbar, a bank of accumulator buffers, and a post-processing unit for ReLU and compression. The ReLU and compression unit in GSCN is more straightforward than SCNN since GSCN doesn't need to communicate with other PEs for the tile edges (Figure 6.6 ③). We expect that

predictor enables us to skip a considerable portion of unused products. Since about 60% of products are not useful in the gradients of weight convolution, ideally, we would get about $2\times$ speedups.

Chapter 7

Evaluation

7.1 Evaluation

In this section, we present 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.

7.2 Experimental Setup

We implement the ReSprop and W-ReSprop algorithms in PyTorch [74]. We use the custom C++ and Cuda extensions of pytorch to write our custom convolution layer. Note that this code is showing the functionality of ReSprop algorithm, and due to the fine-grained sparsity to gain the speedup, a hardware accelerator is required. To evaluate our algorithms, we train three different widely used state-of-the-art architectures; ResNet-18, 34, 50 [32], Wide Residual Networks [96], and VGG-16 [86] on three different datasets: CIFAR-10, CIFAR-100 [47] and ImageNet ILSVRC2012 [82]. 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 reuse. 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 above choice of hyper-parameters follows [32, 36]. For all evaluations in this section, we use the above setup, except in Section 7.4, where we study batch size impact and effect of the number of compute nodes on accuracy.

To model the performance of the GSCN and the baseline (SCNN) architecture, we rely primarily on a custom-built DNNsim cycle-level simulator [22]. We extend this simulator to support GSCN. The simulator is driven by the 90% sparse hybrid gradient, sparse input activation and weights extracted from the PyTorch framework [74] while training with ReSporp. The simulator executes each layer of the network one at a time. We demonstrate the results for GSCN, computing back-propagation convolutions for a mini-batch size of 64 compared to GTX 1080 Ti GPU. The GPU back-propagation convolutions is measured by isolating the back-propagation convolution function in C++ back-end in PyTorch framework.

7.3 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.

7.3.1 Accuracy on CIFAR10 and CIFAR100:

Tables 7.1 and 7.2 show 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.

		CIFAR-100		
RS	**Algorithm**	**ResNet34**	**WRN-28-10**	**VGG-16**
50%	ReSprop	76.02 ± 0.15	81.45 ± 0.17	72.58 ± 0.23
	W-ReSprop	76.4 ± 0.11	81.78 ± 0.16	72.79 ± 0.21
60%	ReSprop	75.81 ± 0.15	80.44 ± 0.16	70.89 ± 0.22
	W-ReSprop	76.01 ± 0.12	81.34 ± 0.15	72.45 ± 0.22
70%	ReSprop	73.92 ± 0.18	78.34 ± 0.11	69.76 ± 0.19
	W-ReSprop	75.60 ± 0.13	81.09 ± 0.15	71.98 ± 0.23
80%	ReSprop	70.76 ± 0.15	76.87 ± 0.13	66.04 ± 0.29
	W-ReSprop	75.44 ± 0.17	80.87 ± 0.14	71.88 ± 0.23
90%	ReSprop	69.12 ± 0.13	75.06 ± 0.10	65.32 ± 0.21
	W-ReSprop	$\mathbf{75.14} \pm 0.16$	$\mathbf{80.38} \pm 0.17$	$\mathbf{71.57} \pm 0.24$
Baseline		75.61 ± 0.16	81.29 ± 0.17	72.50 ± 0.21

Table 7.1: Validation accuracy of ReSprop and W-ReSprop at different reuse-sparsity constraints on the CIFAR-100.

7.3.2 Accuracy on ImageNet:

Table 7.3 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 evaluate W-meProp, a variation of the meProp algorithm employing a warm up phase. Figure 7.1 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 loss of accuracy for meProp and W-meProp.

		CIFAR-10		
RS	**Algorithm**	**ResNet34**	**WRN-28-10**	**VGG-16**
50%	ReSprop	95.85 ± 0.06	96.58 ± 0.09	93.35 ± 0.18
	W-ReSprop	95.91 ± 0.05	96.93 ± 0.11	93.28 ± 0.19
60%	ReSprop	95.25 ± 0.04	95.89 ± 0.11	93.18 ± 0.14
	W-ReSprop	95.41 ± 0.09	96.79 ± 0.07	93.26 ± 0.15
70%	ReSprop	95.01 ± 0.07	95.68 ± 0.08	92.63 ± 0.16
	W-ReSprop	95.23 ± 0.09	96.13 ± 0.15	92.91 ± 0.17
80%	ReSprop	94.17 ± 0.07	93.23 ± 0.08	91.90 ± 0.18
	W-ReSprop	94.96 ± 0.13	95.93 ± 0.12	92.64 ± 0.17
90%	ReSprop	91.61 ± 0.09	90.71 ± 0.15	90.01 ± 0.18
	W-ReSprop	$\mathbf{94.36} \pm 0.07$	$\mathbf{95.67} \pm 0.11$	$\mathbf{92.43} \pm 0.18$
Baseline		95.13 ± 0.09	96.30 ± 0.11	93.25 ± 0.15

Table 7.2: Validation accuracy of ReSprop and W-ReSprop at different reuse-sparsity constraints on the CIFAR-10.

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

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.

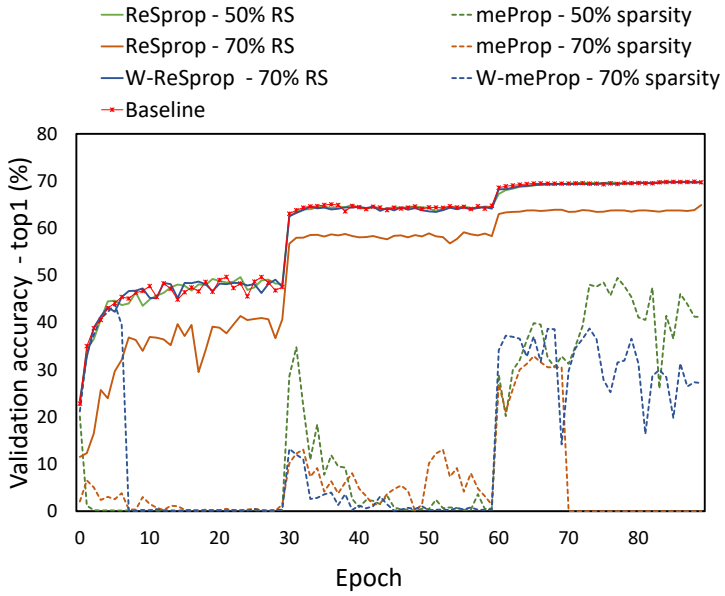


Figure 7.1: Top 1 validation accuracy of ReSprop, W-ReSprop, meProp and W-meProp algorithms for training ResNet-18 on the ImageNet dataset. The baseline is trained with no sparsity or reusing.

7.4 Sensitivity Study

In this section, we analyze the depth and width of the network on ReSprop, we vary the batch size from 32 to 128 while training ResNet-34 on the CIFAR-100 dataset on a single GPU, and train ResNet-18 on the ImageNet dataset with 2,4 and 8 GPUs.

7.4.1 Deep and Wide Networks

Previous studies have shown that network depth and width affect network convergence [60, 77, 84]. Here, we study the effect of depth and width on the ReSprop algorithms. Table 7.4 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 accu-

racy loss of 0.17% for ResNet-50. Similarly, the results for WRN-28-10 shown in Table 7.1 and 7.2, 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 7.4: Validation accuracy of ResNet-18, 34 and 50 on the CIFAR-100 dataset at 90% reuse-sparsity.

7.4.2 Impact of Batch Size:

Here, we explore effects of batch size on the accuracy of ReSprop. Table 7.5 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.

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

7.4.3 Distribute Training Across Multiple Compute Nodes

Data parallelism is a popular way to accelerate training [46, 78]. To explore the impact of distributed training on accuracy, and still ignoring speedup, we evaluate ReSprop on 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 7.6 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 7.6: Top 1 validation accuracy of ResNet-18 on the ImageNet dataset trained on 2, 4 and 8 nodes.

7.5 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.

7.5.1 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 (4.5)) 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.

7.5.2 Pre-ReSprop Overhead

As shown in Section 5.1, the ReSprop algorithm consists of pre-ReSprop and back-ReSprop. Pre-ReSprop can be calculated in parallel with the original forward pass convolution. Figure 7.2 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).

7.5.3 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 7.8 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 [5]. Figure 7.3 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× to 3.0× (on average 2.7×) faster using ReSprop. Among sparse training algorithms, DSG sparsifies back-propagation convolutions (Eq. 4.2 and 4.3). Table 7.7 shows the accuracy and speedup of DSG and W-ReSprop. W-ReSprop with the same sparsity percentage achieves higher accuracy

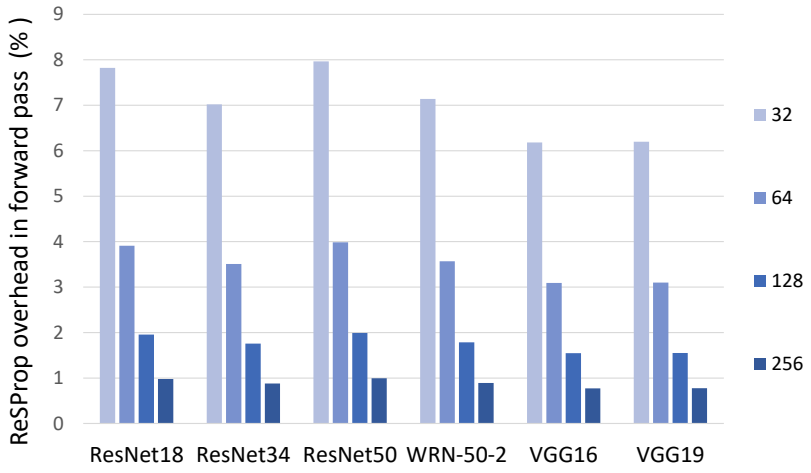


Figure 7.2: Computation overhead of ReSprop at forward pass (pre-ReSprop) for different batch sizes (ImageNet dataset).

and speedup. Reducing dimension for sparsifying gradients and inputs is the main reason for accuracy loss at high sparsities in DSG.

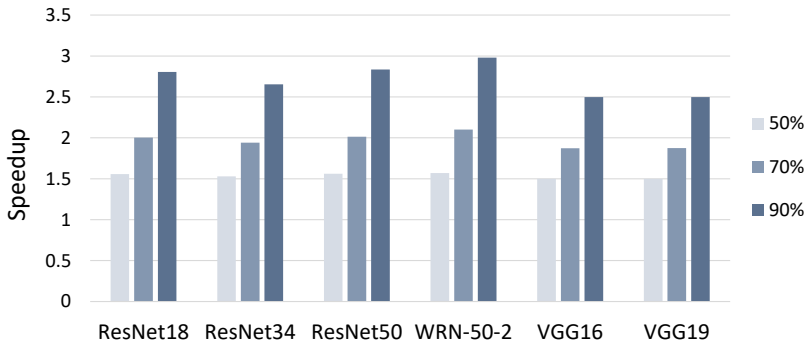


Figure 7.3: ReSprop training (forward+backward) speedup versus architecture for three reuse-sparsity percentages (ImageNet).

	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 7.7: 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 7.8: Theoretical and GSCN speedup at backward pass computations with 90% resue-sparsity (ImageNet).

7.5.4 Accelerator for Sparse Back-propagation:

We modify the SCNN [72] 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 [22]. We extend this simulator to support GSCN.

Table 7.9 lists the key parameters of the GCN design we explore in this paper. The design employs four tiles of 8×8 array of PEs, each PE with a 4×4 cartesian multiplier array, and an accumulator buffer with 32 banks. Since we need a generic architecture, we set the GSCN's buffer size in a way that it is sufficient for all forward and backward pass convolutions. The filter and input buffer each carry a 4-bit overhead for each 16-bit value to encode the compressed coordinates.

To run more than one input sample parallel (mini-batch training) in GSCN design GSCN will tile the PEs in which $n \times n$ PE array is called a tile. Tiles will fetch the same filters (reuse the filters), but each tile has a different input sample. Since PE tiles are applying convolutions on different input samples, there is no need of communicate among them.

Figure 7.4 shows the speedup on GSCN compared to SCNN and GTX 1080 Ti

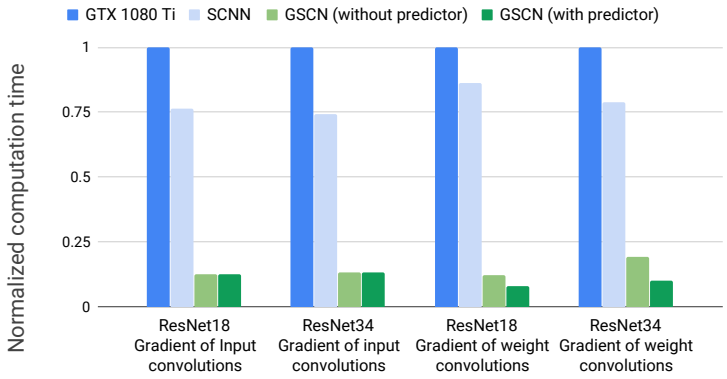


Figure 7.4: Figure shows the speedup for GSCN compared to SCNN and GTX 1080 Ti GPU while training with ReSprop.

Table 7.9: GSCN parameters

GSCN Parameters	Values
Tiles	4
PE/Tile	64

PE Parameters	Value
Multiplier width	16 bits
Accumulator width	24 bits
Multiply array (F×I)	4×4
Accumulator banks	32
Accumulator bank entries	32
Filter buffer	2KB
Input buffer	10KB
Output buffer	10KB

GPU. For GPU timing, the execution time for two back-propagation convolution kernels are measured on a real GTX 1080 Ti GPU. The y-axis shows relative computation time based on GPU execution time. This figure demonstrates that adding predictor has made gradient of weight calculation about $2\times$ faster, as we expected.

Table 7.8 shows the speedup we can gain on GSCN accelerator compared to GTX 1080 ti GPU by running standard training (second row) and ReSprop algo-

rithm (third row) on GSCN.

Chapter 8

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. Storing and reusing a random gradient at each iteration enables us to make the pre-calculation overhead about 2% for batch sizes larger than 128. Moreover, the memory overhead becomes negligible by storing just one random gradient. Thus, pre-ReSprop calculations are negligible and can be done in parallel to forward pass. As shown in Figure 8.1 We proposed W-ReSprop as a variant of ReSprop. W-ReSprop enables

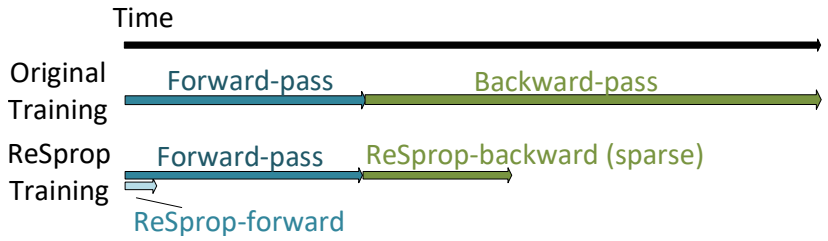


Figure 8.1: ReSprop has two parts. Pre-Resprop computations are negligible and back-ReSprop are sparse computations in backward pass.

us to lose less accuracy compared to ReSprop. W-ReSprop helps network to get adapted to the reuse and after first few epochs we continue the training with the target reuse percentage.

ReSprop and W-ReSprop can be used for training common network architec-

tures and achieves average 2.7× overall speedup in training with negligible loss in model accuracy. The backward pass takes 70% training time, and the speedup gained by ReSprop is due to the 10× faster backward pass. Thus, the overall (theoretical) training speedup can be calculated from the above using Amdahl's Law as:

$$\text{Speedup} = \frac{1}{\underbrace{(1-0.7)}_{\text{forward-pass (not accelerated)}} + \underbrace{\frac{0.7}{10}}_{\text{backward-pass}}}$$

Our experiments show that ReSprop and W-ReSprop are robust to the choice of batch sizes, architectures, and are applicable to distribute training across multiple GPUs.

To find the speedup of ReSprop on a hardware accelerator, we need a sparse accelerator that supports training. Most previous studies on CNN sparse accelerators, however, have paid less attention to the training aspect, despite being more computationally demanding than inference. We modify SCNN accelerator to support sparse back-propagation convolutions to measure the speedup of the ReSprop algorithm on our customized hardware design.

This work introduces a generic sparse convolution neural-network accelerator (GSCN), which overcomes the shortcomings of previous accelerators in back-propagation convolutions. Thus, GSCN can be used both for sparse training and inference. GSCN achieves 6× and 5.5× speed up on ResNet18 and ResNet34 back-propagation, respectively, compared to sparse convolutional neural network (SCNN) accelerator. It also achieves 8.7× and 7.1× speedup compared to the GTX 1080 Ti GPU.

8.0.1 Discussion

ReSprop method can be considered as a new variant of momentum for each individual output gradient parameter. While momentum has a hyperparameter β for the gradient of weights, our method has a fixed $\beta = 1$ or $\beta = 0$ for each output gradient element. Moreover, ReSprop is an orthogonal approach to the choice of optimizers, and it can be used with different optimizers such as Adam or Adagrad.

8.0.2 Future Work

The adaptation of the proposed methodology with a diverse set of CNN architectures, such as fully convolutional networks, could be followed in future work. The future work also includes a deeper analysis of reusing gradient in other non-CNN models such as fully connected and recurrent neural networks. This thesis has been mainly focused on the reusing of output gradient, while analysis of the sparsity of weights combined with the reusing idea could be explored in the future. Also, it could be interesting to consider the effect of reusing gradient on reusing weights and sparsifying the weights.

Moreover, the GSCN accelerator is an ongoing project and many aspects of it such as NOC between PEs, running large batch sizes, and how to gain a higher performance in sparse calculations is an ongoing project.

Bibliography

[1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. in 2016 ieee. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, volume 10, 2016. → pages 3, 6, 26, 27

[2] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394. ACM, 2017. → pages 3, 6, 27

[3] Z. Allen-Zhu and E. Hazan. Variance reduction for faster non-convex optimization. In *International Conference on Machine Learning*, pages 699–707, 2016. → page 23

[4] Z. Allen-Zhu and Y. Yuan. Improved svrg for non-strongly-convex or sum-of-non-convex objectives. In *International conference on machine learning*, pages 1080–1089, 2016. → page 4

[5] G. 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. → pages 5, 59

[6] A. G. Anderson and C. P. Berg. The high-dimensional geometry of binary neural networks. *arXiv preprint arXiv:1705.07199*, 2017. → page 30

[7] S. Anwar, K. Hwang, and W. Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18, 2017. → page 19

[8] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. Scalable methods for 8-bit training of neural networks. In *Advances in Neural Information Processing Systems*, pages 5145–5153, 2018. → page 2

[9] G. Bellec, D. Kappel, W. Maass, and R. Legenstein. Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*, 2017. → page 20

[10] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010. → page 2

[11] L. Bottou and Y. L. Cun. Large scale online learning. In *Advances in neural information processing systems*, pages 217–224, 2004. → pages 15, 22

[12] A. Cauchy. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847. → page 22

[13] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014. → pages 3, 6

[14] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016. → page 25

[15] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016. → page 26

[16] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012. → page 1

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012. → page 2

[18] A. Defazio and L. Bottou. On the ineffectiveness of variance reduced optimization for deep learning. *arXiv preprint arXiv:1812.04529*, 2018. → page 23

[19] A. Defazio, F. Bach, and S. 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. → pages 4, 22, 23

[20] A. Defazio, J. Domke, et al. Finito: A faster, permutable incremental gradient method for big data problems. In *International Conference on Machine Learning*, pages 1125–1133, 2014. → page 22

[21] T. Dettmers and L. Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019. → pages 2, 3, 20

[22] I. Edo, O. Awad, A. H. Zadeh, D. M. Stuart, A. D. Lascorz, M. Nikolić, and A. Moshovos. DNNsim: Deep Learning Accelerators Toolkit. https://github.com/isakedo/DNNsim. → pages 53, 61

[23] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE, 2012. → page 24

[24] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin. The lottery ticket hypothesis at scale. *arXiv preprint arXiv:1903.01611*, 2019. → pages 19, 20

[25] T. Gale, E. Elsen, and S. Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019. → pages 18, 20

[26] M. Golub, G. Lemieux, and M. Lis. Dropback: Continuous pruning during training. *arXiv preprint arXiv:1806.06949*, 2018. → page 21

[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016. → page 15

[28] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. → page 18

[29] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015. → pages 19, 20

[30] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016. → pages 3, 6, 26, 47

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

[32] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. → pages 12, 13, 52, 53

[33] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision*, pages 1389–1397, 2017. → pages 3, 19

[34] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012. → page 1

[35] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision*, pages 646–661. Springer, 2016. → page 2

[36] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017. → page 53

[37] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. → pages 2, 11

[38] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018. → page 22

[39] H. Ji, L. Song, L. Jiang, H. H. Li, and Y. Chen. Recom: An efficient resistive accelerator for compressed deep neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 237–240. IEEE, 2018. → page 27

[40] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013. → pages 4, 22

[41] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017. → page 25

[42] P. Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1(2):139–159, 2009. → page 30

[43] E. D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990. → page 18

[44] A. Katharopoulos and F. Fleuret. Biased importance sampling for deep neural network training. *arXiv preprint arXiv:1706.00043*, 2017. → page 2

[45] A. Katharopoulos and F. Fleuret. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942*, 2018. → page 2

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. → pages 1, 2, 10, 57

[47] A. Krizhevsky et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. → page 52

[48] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605, 1990. → page 18

[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998. → page 15

[50] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012. → page 15

[51] N. Lee, T. Ajanthan, and P. H. Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018. → page 19

[52] N. Lee, T. Ajanthan, S. Gould, and P. H. Torr. A signal propagation perspective for pruning neural networks at initialization. *arXiv preprint arXiv:1906.06307*, 2019. → page 18

[53] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. → page 19

[54] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie. Dynamic sparse graph for efficient deep learning. *arXiv preprint arXiv:1810.00859*, 2018. → pages 2, 3, 20

[55] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405. IEEE, 2016. → page 25

[56] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, et al. Reno: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015. → page 24

[57] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. → page 20

[58] C. Louizos, K. Ullrich, and M. Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017. → page 3

[59] C. Louizos, M. Welling, and D. P. Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017. → page 3

[60] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239, 2017. → page 56

[61] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *IEEE International Conference on Computer Vision*, pages 5058–5066, 2017. → page 3

[62] M. Mahdavi, L. Zhang, and R. Jin. Mixed optimization for smooth functions. In *Advances in neural information processing systems*, pages 674–682, 2013. → page 4

[63] J. Mairal. Incremental majorization-minimization optimization with application to large-scale machine learning. *SIAM Journal on Optimization*, 25(2):829–855, 2015. → page 22

[64] H. Malmgren, M. Borga, and L. Niklasson. *Artificial Neural Networks in Medicine and Biology: Proceedings of the ANNIMAB-1 Conference, Göteborg, Sweden, 13–16 May 2000*. Springer Science & Business Media, 2012. → page 1

[65] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9 (1):2383, 2018. → page 20

[66] D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. JMLR. org, 2017. → page 3

[67] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016. → page 18

[68] J. Morajda. Neural networks and their economic applications. In *Artificial intelligence and security in computing systems*, pages 53–62. Springer, 2003. → page 1

[69] H. Mostafa and X. Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. *arXiv preprint arXiv:1902.05967*, 2019. → pages 3, 20

[70] S. Narang, E. Elsen, G. Diamos, and S. Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017. → page 37

[71] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013. → page 22

[72] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. 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. → pages xii, 3, 6, 27, 39, 43, 61

[73] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013. → page 13

[74] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017. → pages 52, 53

[75] J. L. Patel and R. K. Goyal. Applications of artificial neural networks in medical science. *Current clinical pharmacology*, 2(3):217–226, 2007. → page 1

[76] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. → page 1

[77] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein. On the expressive power of deep neural networks. In *International Conference on Machine Learning*, pages 2847–2854. JMLR. org, 2017. → page 56

[78] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *International Conference on Machine Learning*, pages 873–880. ACM, 2009. → page 57

[79] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, and A. Smola. Stochastic variance reduction for nonconvex optimization. In *International Conference on Machine Learning*, pages 314–323, 2016. → page 23

[80] N. L. Roux, M. Schmidt, and F. R. Bach. A stochastic gradient method with an exponential convergence _rate for finite training sets. In *Advances in neural information processing systems*, pages 2663–2671, 2012. → page 22

[81] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. → page 14

[82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3): 211–252, 2015. → pages 4, 52

[83] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 304–317, 2019. → page 26

[84] O. Sharir and A. Shashua. On the expressive power of overlapping operations of deep networks. *arXiv preprint arXiv:1703.02065*, 2017. → page 56

[85] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *Advances in Neural Information Processing Systems*, pages 568–576, 2014. → page 1

[86] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. → pages 12, 52

[87] R. Spring and A. 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. → page 2

[88] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014. → page 11

[89] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. → page 11

[90] X. Sun, X. Ren, S. Ma, and H. 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. → pages 2, 3, 7, 20, 28, 31

[91] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013. → page 16

[92] R. W. Vuduc and J. W. Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003. → page 47

[93] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, pages 7675–7684, 2018. → page 2

[94] B. Wei, X. Sun, X. Ren, and J. Xu. Minimal effort back propagation for convolutional neural networks. *arXiv preprint arXiv:1709.05804*, 2017. → pages 2, 3, 7, 20

[95] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016. → page 19

[96] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. → pages 12, 14, 52

[97] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016. → page 27

[98] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017. → page 2

[99] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28. IEEE, 2018. → page 27

[100] M. Zhu and S. Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017. → page 37