MEMORY EFFICIENT FILTERING ALGORITHMS FOR CONVOLUTIONAL NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

By
Bahadır Alp Çakır
December 2020

MEMORY EFFICIENT FILTERING ALGORITHMS FOR CONVOLUTIONAL NEURAL NETWORKS

By Bahadır Alp Çakır

December 2020

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ömer Morgül(Advisor)

Tolga Çukur

Mehmet Önder Efe

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

MEMORY EFFICIENT FILTERING ALGORITHMS FOR CONVOLUTIONAL NEURAL NETWORKS

Bahadır Alp Çakır
M.S. in Electrical and Electronics Engineering
Advisor: Ömer Morgül
December 2020

Deployment of state of the art CNN architectures like Xception, ResNet and GoogleNet in resource limited devices is a big challenge. These architectures consist of many layers and millions of parameters. Moreover, they require billions of floating point operations to inference just an image. Therefore, memory space needed to store parameters and to execute them are the main constraints for efficient convolutional neural network architectures.

In this thesis, we examine Winograd's minimal filtering algorithms to reduce number of floating point operations performed in convolutional layers. We reduce the number of multiplications x2.25 times without any accuracy loss. Moreover, we investigate, sparse and quantized Winograd's algorithms so that we can make conventional Winograd algorithms more memory efficient. We propose a linear quantization scheme to quantize weights of the networks more than 1-bit. We use ReLU activation function and Targeted Dropout which is a variant of Dropout to prune transformed inputs of Winograd algorithm. We binarize weights so that most arithmetic operations are converted to bit-wise operations. We conduct several experiments on CIFAR10 and CIFAR100 datasets and discuss the classification performances of both conventional and modified Winograd minimal filtering algorithms. We achieve less than 1.9% classification error with ReLU-ed Winograd CNN compared to conventional Winograd. We reduce memory requirements up to x32 times by binarizing weights of ReLU-ed Winograd CNN, and in return we incur around 2% accuracy loss. Lastly, for applications which are less tolerant to accuracy loss, rather than binarizing weights we quantize them to 2-bit, 4-bit and 8-bit. Our quantized ReLU-ed Winograd CNNs reach same accuracy levels as ReLU-ed Winograd CNN.

Keywords: Winograd's minimal filtering algorithms, ReLU, targeted dropout, binary weights, quantized weights, memory efficiency.

ÖZET

EVRİŞİMLİ YAPAY SİNİR AĞLARI İÇİN BELLEK VERİMLİ FİLTRELEME ALGORİTMALARI

Bahadır Alp Çakır
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Ömer Morgül
Aralık 2020

Xception, ResNet ve GoogleNet gibi en son tasarlanan evrişimli yapay sinir ağı mimarilerinin, kaynak sınırlı cihazlarda kullanılması büyük bir uygulama problemidir. Bu mimariler, fazla sayıda katmandan ve milyonlarca parametreden oluşturmaktadır.Ayrıca, bunlar sadece bir resmi işleyip, sonuçlandırabilmek için milyarlarca kayan nokta operasyonuna ihtiyaç duymaktadır. Dolayısı ile, parametreleri depolamak ve onları kullanmak için gerekli depolama alanları, verimli evrişimli sinir ağı mimarileri için ana kısıtlamalardır.

Bu tez çalışmasında, evrişimli katmanlarda gerçekleştirilen kayan nokta işlemlerinin sayısını azaltmak için Winograd'ın minimal filtreleme algoritmalarını inceliyoruz. Herhangi bir doğruluk kaybı olmaksızın, toplam çarpma işlemi sayısını x2.25 kat azaltıyoruz. Dahası, geleneksel Winograd algoritmalarını depolama alanları açısından daha verimli hale getirmek için, seyrek ve nicelenmiş Winograd algoritmalarını inceliyoruz. Ağ parametrelerini 1 bitten fazla nicelemek için doğrusal bir niceleme şeması öneriyoruz. ReLU aktivasyon fonksiyonunu ve terkinimin bir varyasyonu olan hedefli terkinimi Winograd algoritmasının dönüştürülmüş girdilerini seyreltmek için kullanıyoruz. Bir çok aritmetik işlemi, bit tabanlı işlemlere dönüştürmek için parametreleri, -1 veya 1, değerlerini alacak şekilde ikilileştiriyoruz. CIFAR10 ve CIFAR100 veri kümeleri üzerinde farklı deneyler yaparak, geleneksel Winograd algoritmaları ile değişiklik yapılan algoritmalar arasında sınıflandırma performanslarını tartışıyoruz. Geleneksel Winograd algoritması ile karşılaştırıldığında ReLU-ed Winograd algoritması %1.9'den daha düşük sınıflandırma performansı kaybı elde edebiliyoruz. ReLU-ed Winograd algoritmasındaki parametreleri ikilileştirerek, bellek ihtiyacını x32 kat azaltabiliyoruz, karşılığında %2'den daha az sınıflandırma performans kaybı ile karşılaşıyoruz. Son olarak, sınıflandırma performans kaybına

karşı düşük toleranslı uygulamalar için parametreleri ikilileştirmek yerine onları 2-bit, 4-bit ve 8-bit olacak şekilde niceliyoruz. Nicelenen parametreler ile eğitilen ReLU-ed Winograd evrişimli sinir ağlarının, nicelenmeyen parametreler ile eğitilen ağ ile benzer sınıflandırma performansına eriştiğini gözlemliyoruz.

Anahtar sözcükler: Winograd'ın minimal filtreleme algoritmaları, ReLU, hedefli terkinim, ikilileştirmiş parametreler, niceleştirilmiş parametreler, bellek verimliliği.

Acknowledgement

I would like to express my sincere gratitude to my advisor Ömer Morgül for his endless support, understanding and patience. He guided me through our research and understood my late responses. Without him, it wouldn't be possible to write this thesis.

I would like to thank Tolga Çukur and Mehmet Önder Efe for being part of my committee members and their valuable comments.

I also acknowledge Aselsan Inc. and my colleagues for given me an opportunity to work in sincere and friendly working environment. They supported me a lot throughout my studies.

I wish to thank my family with my heartfelt gratitude. They always stood behind me and supported me in every decision I made. Their guidance light my way and make me who I am. Without them, my achievements wouldn't be possible.

Last but not least, I want to thank my fiancee, soon to be my wife, Gizem. She is always there for me to support and motivate during my hard times. She brings happiness and peace to my life. Without her, the world would be a dark and cheerless place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep convolutional neural networks move state of the art forward on many computer vision tasks such as object detection, image classification and face recognition [4], [5], [6]. Modern graphical processing units, GPUs, large datasets and latest designed architectures provide us to develop more deeper convolutional neural network, CNN, models. Ever since AlexNet wins Imagenet classification challenge, there is a trend to built more deep CNNs to get higher accuracy. Today, challenge winner models have more than 100 layers, where AlexNet has just 8. [7]

Although more deeper CNNs achieve high classification performance, they are power-hungry. They have millions of parameters and require millions of floating point operations. For example, ResNet has above 50 million parameters and for a colorful image with size 224x224, needs farther than 10 Giga floating point operations [7]. These can not be afforded by power limited devices like mobile phones, tablets, Internet of Things, IoT, devices etc.

There are certain constraints to deploy deep CNN models in resourse limited devices. These constraints can be associated with 3 categories: model size, runtime memory and number of floating point operations. Model size: models with deep network architectures require their structure and parameters to be stored in

huge memories. For ResNet, memory space needed to store 32-bit floating point weights is around 168 MB, which is more than an embedded device can afford. Run-time memory: for inference time, memory needed to execute CNN models can require more memory space than to store them. For a minibatch of 32 images, ResNet needs over 7 GB memory space from local DRAM. Most typical RAM size for mobile devices is between 1 GB and 2GB, so it is unfeasible to deploy trained state of art CNN models on mobile phones. Floating point operations: by its name CNNs run convolutions over high resolution images. Depending on filter size, millions of multiplication and addition operations should be performed. Even on computers with powerful GPUs, these operations take much time, for example on a computer with a NVIDIA GTX 1660 TI GPU, processing a minibatch of 32 images with 32x32 resolution over pretrained CNN models takes around 100 ms, which could take hours for embedded devices. Many studies have been proposed to make large CNN models more efficient or train computation efficient CNN models for resource limited devices. Gong et al. [8], shows that by applying vector quantization methods, parameters of large CNNs can be compressed up to 24 times while Top-5 accuracy on Imagenet Large Scale Visual Recognition 2012,ILSVRC2012, decreases less than %1. HashedNets [9], use a hash function to group weigths randomly into hash buckets and force all weights in the buckets to share same value. So, rather than storing whole parameter set, values for each bucket is stored. Han et al. [10] proposes deep compression which is a process to compress large scale CNNs with pruning, quantization and Huffman coding sequentially. Connections with lowest magnitude are pruned from the network and remaining 32-bit float parameters represented as 5-bit fixed numbers. Deep compression reduces the memory sizes required to store Alexnet and VGG16 which is a deep convolutioal neural network with 16 layers proposed by Visual Geometry Group, VGG, [11], x39 and x49 times respectively without accuracy loss. In [12], [13] comparably small and computationally efficient CNN architectures are proposed for embedded devices. These architectures use depth-wise separable convolutions instead of direct convolutions. Basically, depth-wise separable convolution divides direct convolution into two convolutions: a separable convolution followed by a point-wise convolution. Separable convolution is a type of grouped convolution, it divides input and filter into channel-wise groups. Each

group is multiplied with each other and then results are combined over channels with point-wise convolution. Point-wise convolutions are actually direct convolutions with 1x1 kernel size. For 3x3 kernels, depth-wise separable convolutions require 8 to 9 times less memory space and less computations compared to direct convolutions [12].

Researches show that high precision parameters are not essential to achieve high classification performance. More redundant and computationally efficient CNNs can be built with low precision parameters. Models with low precision parameters require less memory space and less arithmetic operations compared to full precision networks. [14] proposes stochastic rounding to quantize 32-bit floating point weights to 16-bit fixed point weights and incurs almost no loss in classification performance. [15], [16] and [17] propose different quantization schemes and represent floating point activations with very low bitwidths as well as weights. They show that quantized networks achieve same accuracy levels as well as full precision network on larger dataset like Imagenet. BinaryConnect [18] constraints weights to be -1 or 1 during feed-forward phase by expectation backpropagation (EBP) proposed in [19]. 32-bit floating point operations are converted to bit-wise operations and memory space required for inference drastically reduced by 32 times. BinaryConnect achieves state of the art accuracy levels on small datasets like MNIST[20] and CIFAR10. Binarized Neural Networks[21] and XNOR-Net[22] use similar approach to binarize activations and gradients as well.

Deep CNN models are generally over-parameterized and contain much redundancy in them [23]. There are parameters which have almost no contribution to overall performance. Contrary, they increase the need for memory space both in run-time and for storage of the network. From early stages of deep neural networks, several researches have been done to remove redundant parameters from the models. Optimal Brain Damage [24] and Optimal Brain Surgeon [25] use second order Taylor series expansion to evaluate weights for pruning. [26] and [27] propose magnitude based pruning strategies to prune weights. In these works, the parameters with small magnitudes are treated as redundant variables. [28] and [29] also use magnitude based pruning strategies to prune activation units in training time.

Convolutional layers are the major part of CNNs, they use direct convolution to combine filter and input values. Direct convolution comprised of multiply-accumulate (MAC) operations. While training, multiplication operations require so much power and occupy much memory space compared to other arithmetic operations. In recent years, a considerable amount of research is devoted to change algebraic property of convolution or optimize direct convolution. [30] and [31] use Fast Fourier Transforms,FFTs, to reduce arithmetic complexity for convolutional layer. [2] reduces multiplications performed for direct convolutions up to 4 times by using minimal filtering algorithms, which are implemented on cuDNN which is CUDA Deep Neural Network library for GPU training, to increase computational efficiency.

In this thesis, we investigate Winograd's minimal filtering algorithms and other techniques proposed in literature to built energy friendly, efficient convolutional neural networks. We use Winograd's minimal filtering algorithms defined in [2] to correlate input and filters for the convolutional layers of our architecture. We apply certain modifications to increase computational efficiency. We study the low precision arithmetics by quantizing weights in our architecture. Moreover, redundant activation units are removed by applying ReLU and Targeted Dropout [29] to transformed inputs.

Main contributions of this thesis as follows:

- Filter transformation are excluded from Winograd's minimal filtering algorithms. Moreover, we initialize kernels with transformed kernel size.
- A linear quantization scheme which empirically achieves state of the art results is given for the networks whose parameters are uniformly distributed and normalized.

1.1 Outline of Thesis

In Chapter 2, direct convolution operation and convolutional neural networks are introduced. Winograd's minimal filtering algorithms and number of multiplications to execute them are compared with direct convolution. Details of algorithm as well as arithmetic complexity of the algorithm are examined in detail.

In Chapter 3, binarization and quantization scheme for network parameters used for low precision arithmetics are introduced . A linear quantization scheme to represent real-valued parameters to low precision parameters is proposed. Gradient computation and memory efficiency details of quantized networks are given.

In Chapter 4, concept of pruning and dynamic sparsity of activations are defined. ReLU operation and its sparsity effect for activation units is mentioned. Moreover, Targeted Dropout that is a variant of Dropout is introduced. Magnitude based pruning techniques and their usages for weights and units of a network is discussed.

In Chapter 5, conducted simulations and implementation details of the algorithms are given. Modifications over conventional Winograd algorithm are detailed. Hyperparameters, training setup and datasets used to train and validate CNN models are given. Test accuracy results for different models are reported.

In Chapter 6, simulation results and trained architectures are discussed. Advantage and disadvantage of conventional Winograd algorithm and other modified algorithms are mentioned.Finally, in Chapter 7, conclusion and future works for further investigation are given.

Chapter 2

Convolutional Neural Networks and Fast Convolution Algorithms

2.1 Neural Networks

A neural network is a sequence of arithmetic operations to learn relationships in a set of input data with a process that mimics the behavior of neuron activities in brain. A neuron, collects the information from data and classifies it. Basically, data coming from different variables are weighted and accumulated in a neuron. Then, the accumulated data activated by a non-linear activation function. Figure 2.1 shows an example of single neuron activity which can be given as,

$$\widehat{a} = f\big(\sum x_1 w_1 + x_2 w_2 + b\big) \tag{2.1}$$

where $f(\cdot)$ is non-linear function, x_1, x_2 input data, w_1, w_2 weights and b is bias term.

Neural networks consist of layers and connections between them. A layer contains one or more artificial or organic neurons or so called nodes. A neural network with a single hidden layer can be shown as Figure 2.2 which are called

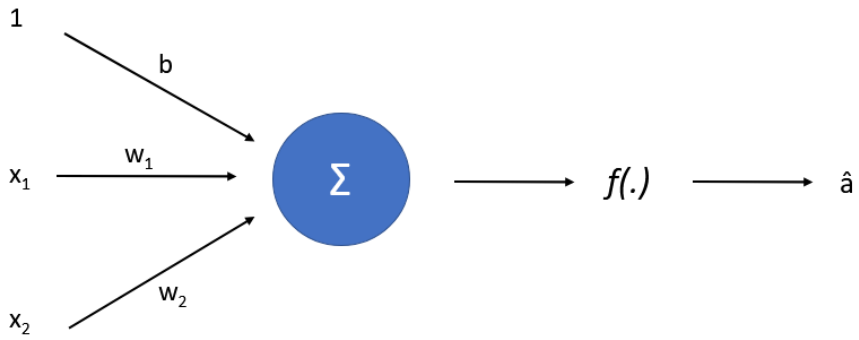


Figure 2.1: Single neuron behavior for input data x_1, x_2 and weights w_1, w_2 with a bias term, b.

fully-connected layers. By adding more hidden layers we can built more deep architectures which are called deep neural networks. As we go deeper, we can learn non-linear relations between input variables, better.

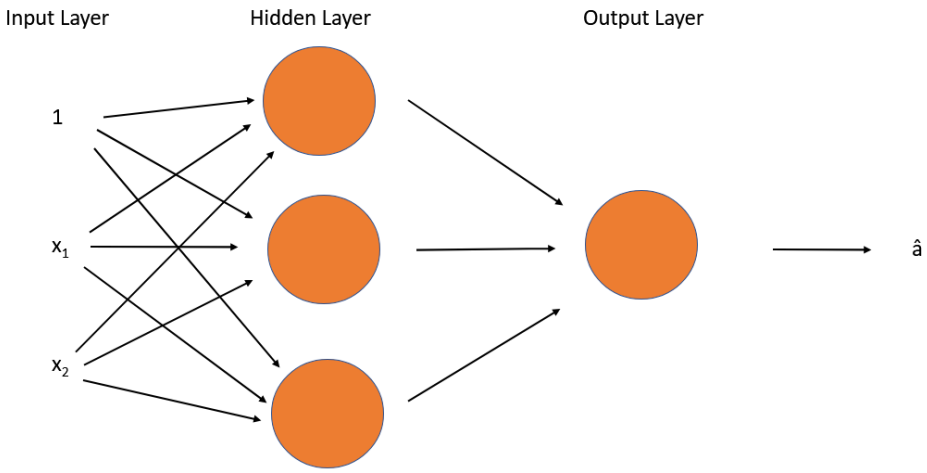


Figure 2.2: A neural network with a single hidden layer.

Training a neural network is an iterative process, which learns the features of input variables with respect to resulting activations. In each iteration, resulting activations of output layer is compared with real outcomes and for each output, an error is calculated with respect to a given error function so called loss function [32]. Gradients of loss function are calculated with respect to weights and inputs, then calculated error backpropagated to weights and inputs of the network over these gradients. For further details, refer to [33], [34].

2.2 Convolutional Neural Network

Convolutional neural networks, also known as CNNs, are types of neural networks which are in general, comprised of convolutional layers, pooling layers and fully-connected layers. An example CNN architecture is given in Figure 2.3. They are mainly used in computer vision tasks. By its name convolutional layers are the major parts of CNNs. A convolutional layer basically convolves N images with C channels and K filters with C channels. [2]. Resulting output Y can be defined as feature map or activation map. If we define, a kernel as $W_{k,c}$ and a image tile as $X_{i,c}$, the output of single convolutional layer can be written as:

$$Y_{i,k} = \sum_{1}^{C} X_{i,c} * W_{k,c} \tag{2.2}$$

where $*$ represents 2D convolution. Here, an image tile is a subset of image containing consecutive pixels.

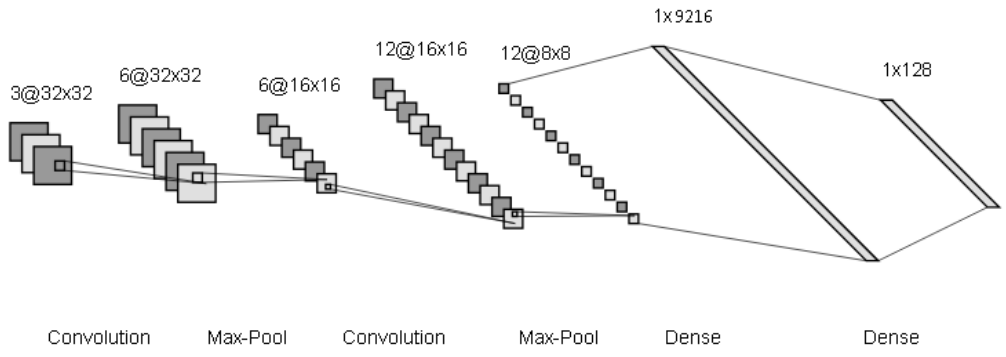


Figure 2.3: Convolutional Neural Network architecture for a colorful input image with 32x32 resolution which is represented as $3@32 \times 32$.

Convolutional layers extract the features of an image by filtering them with fixed sized filters. These filters are the feature detectors. Depending on task, their values can be constant or learned by training. To learn weights of a filter, we can use backpropagation. For a convolutional layer, gradients of input with respect

to loss function is a convolution of 180° flipped filters and backpropagated error of previous layer. Gradients of weights with respect to loss is also a convolution of input and backpropogated error [2]. This is illustrated in Figure 2.4.

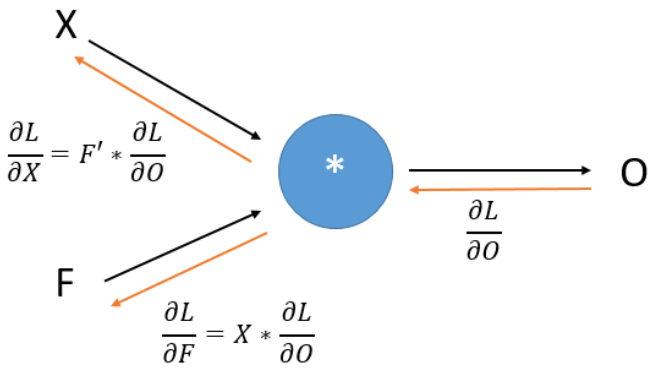


Figure 2.4: Gradients of input, $\frac{\partial L}{\partial X}$ and filter $\frac{\partial L}{\partial F}$, with respect loss function, L, are given. F' shows 180° flipped filters, $*$ represent 2D convolution and O is the output of convolution of input X and F.

Pooling layers are generally inserted between consecutive convolutitonal layers to reduce spatial dimensions of inputs of convolutional layers. In general, they operate in the form of 2×2 filter with a stride of 2 which downsamples height and width of input by 2 for each input channels.Max-pooling layers take maximum value in 2×2 region and omit the other values.

To perform 2D convolution for an image with size (C_{in}, H, W), each channel of input images is divided into $r \times r$ tiles where r is the kernel size. Then, (C_{in}, r, r) image tiles are converted to column matrices, $(N, (C_{in} \times r \times r))$. Each filter of convolutional layer is also converted to column matrices, $(C_{out}, (C_{in} \times r \times r))$. So, whole convolution operation can be executed as 2D general matrix multiplication(GEMM). Finally, columns of output matrix are converted to original size, (C_{in}, H, W) for each row of that matrix. This operation is illustrated in Figure 2.5.

Therefore, number of multiplications, L, performed for 2D convolution of N

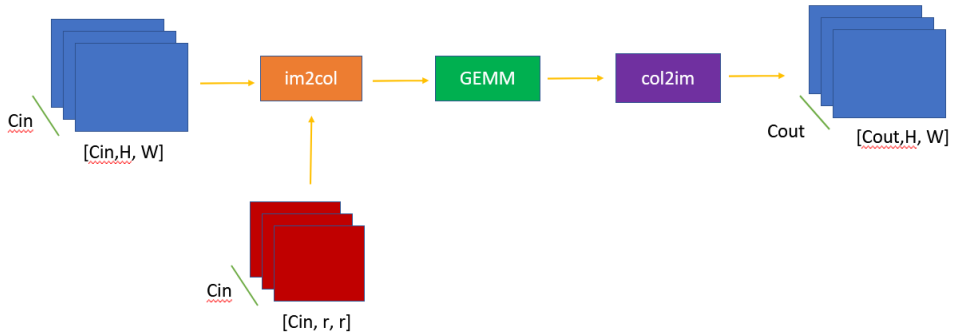


Figure 2.5: Standard convolution operation for an image with size (C_{in}, H, W) and a filter with size (C_{in}, r, r).

images can be computed as,

$$L = r^2 NHWC_{in}C_{out} \tag{2.3}$$

Here r is the kernel size, N is the number of images in minibatch, H and W represent height and width of an image, C_{in} and C_{out} are the number of input and channels respectively.

In most CNN architectures, rather than one large receptive field convolutional layer, stacking convolutional layers with small size filter is prefered. This is because as opposed to large size filters, stacked small filters contain much non-linearities that make their features more expressive. Also, small size filters reduce computational cost. If we think C channels for each volume, number of multiplications required for 7x7 kernel size layer is $7 \times 7 \times C \times C = 49C^2$ whereas 3 stacked convolution layers with 3x3 kernel size require $3 \times (3 \times 3 \times C \times C) = 27C^2$ [35]. So, like most CNN architectures, the arhitectures which are implemented in this thesis use 3x3 kernel size for convolutional layers.

2.3 Convolution with Winograd Minimal Filtering Algorithms

Winograd' s minimal filtering algorithms propose a fast convolution method to convolve small size filters over small tiles with minimal complexity [2]. Most arithmetic operations are carried out by matrix multiplications which reduces number of multiplications up to 4 compared to direct convolution, depending on image tiles.

1D Minimal filtering algorithms are represented as $F(m, r)$ where m is the number of outputs and r is the size of the filter. The number of multiplications, $\mu(\odot)$ required to compute $F(m, r)$ is formulated in [36] as:

$$\mu(F(m, r)) = m + r - 1 \tag{2.4}$$

Filtering an input vector, $X = [x_0, x_1, x_2, x_3]$ with a filter, $W = [w_0, w_1, w_3]^T$ where resulting output is $O = [o_1, o_2]^T$, can be given as follows,

$$X * W = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \begin{bmatrix} x_0 * w_0 + x_1 * w_1 + x_2 * w_2 \\ x_1 * w_0 + x_2 * w_1 + x_3 * w_2 \end{bmatrix} \tag{2.5}$$

where $*$ represent 1D convolution. Here input vector X is divided into 1x3 subsets, $s_1 = [x_0, x_1, x_2]$ and $s_2 = [x_1, x_2, x_3]$ and these subsets are converted into row matrices, then multiplied with filter, W. Number of multiplications required for this operation is 6. We can reduce the number of multiplications by redefining output values $O = [o_1, o_2]^T$ as suggested in [36],

$$F(2, 3) = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \tag{2.6}$$

where,

$$m_1 = (x_0 - x_1)w_0 \qquad m_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2}$$
$$m_4 = (x_1 - x_3)w_2 \qquad m_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2}$$

As we notice, the algorithm uses 4 multiplications therefore it is minimal by the formula given in (2.4). However, other than 4 multiplications, it has additional arithmetic operations which are called transformation operations. The algorithm performs 4 additions to transform input values, 3 additions ($w_0 + w_2$ counted once) and 2 multiplications by a constant to transform filter values and again 4 additions to transform output values. So, we can reformulate (2.6) in matrix form as follows,

$$Y = A^T[(GW) \odot (B^T X)] \tag{2.7}$$

where $\odot$ denotes element-wise multiplication and A^T, G, B^T are the transformation matrices which are given below [2].

$$\begin{aligned} B &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \\ G &= \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \\ A^T &= \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \end{aligned} \tag{2.8}$$

By nesting 1D algorithm $F(m, r)$ with itself, we can form 2D minimal algorithm $F(m \times m, r \times r)$ whose resulting output is formulated as,

$$Y = A^T[(GWG^T) \odot (B^T XB)]A \tag{2.9}$$

where W is a 2D kernel with size $r \times r$ and X is an image tile with size $(m+r-1) \times (m+r-1)$. These algorithms require,

$$\begin{aligned} \mu(F(m \times m, r \times r)) &= \mu(F(m,r))\mu(F(m,r)) \\ &= (m+r-1)(m+r-1) \end{aligned} \tag{2.10}$$

multiplications [37].

Assume X is a single channel image with 4x4 resolution and W is a single channel filter with 3x3 kernel,

$$\begin{aligned} X &= \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix} \\ W &= \begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix} \end{aligned} \tag{2.11}$$

We can write $F(2 \times 2, 3 \times 3)$ as,

$$\begin{aligned} F(2 \times 2, 3 \times 3) &= \begin{bmatrix} x_0 & x_1 & x_2 & x_4 & x_5 & x_6 & x_8 & x_9 & x_{10} \\ x_1 & x_2 & x_3 & x_5 & x_6 & x_7 & x_9 & x_{10} & x_{11} \\ x_4 & x_5 & x_6 & x_8 & x_9 & x_{10} & x_{12} & x_{13} & x_{14} \\ x_5 & x_6 & x_7 & x_9 & x_{10} & x_{11} & x_{13} & x_{14} & x_{15} \end{bmatrix} \times \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{bmatrix} = \begin{bmatrix} o_1 \\ o_2 \\ o_3 \\ o_4 \end{bmatrix} \\ &= \begin{bmatrix} X_0 & X_1 & X_2 \\ X_1 & X_2 & X_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} O_1 \\ O_2 \end{bmatrix} = \begin{bmatrix} M_1 + M_2 + M_3 \\ M_2 - M_3 - M_4 \end{bmatrix} \end{aligned} \tag{2.12}$$

where,

$$M_1 = (X_0 - X_1)W_0 \qquad M_2 = (X_1 + X_2)\frac{W_0 + W_1 + W_2}{2}$$
$$M_4 = (X_1 - X_3)W_2 \qquad M_3 = (X_2 - X_1)\frac{W_0 - W_1 + W_2}{2}$$

As we notice $F(2 \times 2, 3 \times 3)$ uses $4 \times 4 = 16$ multiplications to compute output whereas standard convolution algorithm uses $2 \times 2 \times 3 \times 3 = 36$ which is an x2.25 arithmetic complexity reduction. Moreover, it performs 32 additions for input transform, 28 floating point operations containing additions and multiplications with a constant, for filter transform and 32 additions to reduce multiplication results to final result, which is called inverse transform [2].

2.3.1 Forward Propagation with F(2x2,3x3)

Activations of convolutional layer with kernel size $r \times r$ can be computed by $F(m \times m, r \times r)$. To do that a single channel image with resolution $H \times W$ will be divided into $P = (H/m)(W/m)$ tiles for each channel with a stride $r - 1$, here we assume both (H/m) and W/m have no remainders. And, outputs of each tile and filter multiplication are accumulated. In Figure 5.2a, a conventional algorithm for $F(2 \times 2, 3 \times 3)$ is given for a single image tile, where inputs are previous layer's activations.

Unlike direct convolution algorithm, Winograd minimal filtering algorithms use element-wise multiplications to generate outputs, which is a really unefficient way to implement the algorithm on Central Processing Unit, CPU, GPU and Field Programmable Gate Array, FPGA, devices. Since most benchmarks used in deep learning, very optimized and efficient implementations of matrix multiplications, we convert element-wise multiplications defined in Figure 5.2a to matrix multiplications. Today, GEMM can be stated as the heart of convolutional layers so element-wise multiplications are converted into batched GEMM operations in our implementations.

Transformations of input, filter and output values are another challenges to

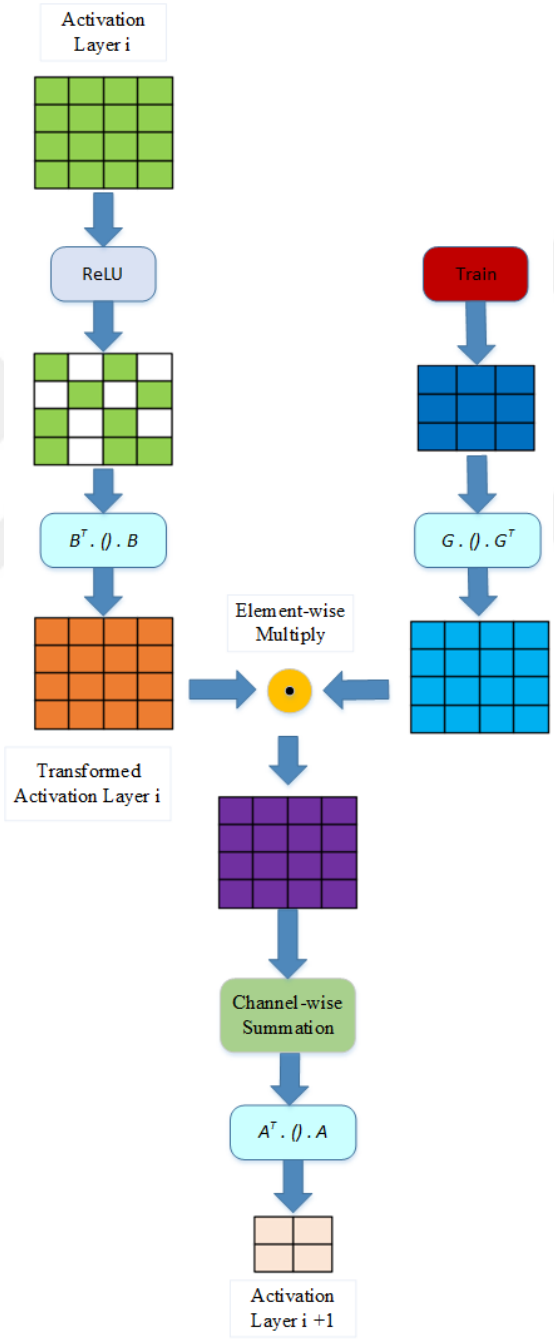


Figure 2.6: Conventional Winograd minimal filtering algorithm for $F(2\times2, 3\times3)$.

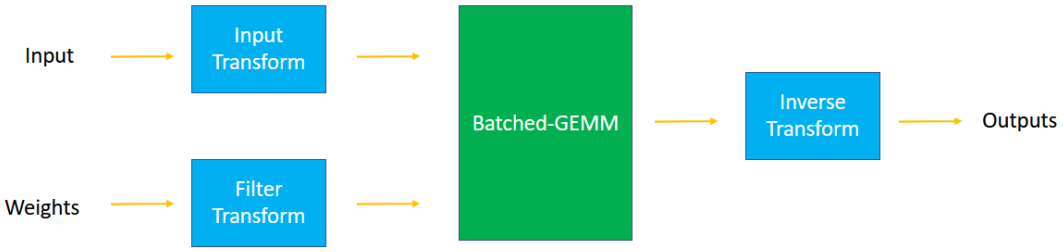


Figure 2.7: Winograd minimal filtering algorithm for convolution of input images with weights.

implement Winograd's minimal filtering algorithm. Their optimization could be a game changer for computational time required to execute algorithm. Although,theoretically, tranformations can be done by addition operations.In practice, we experienced that addition operations can be more costly than matrix multiplications for tensors. So, tranformations of input, filter and output values are implemented as matrix multiplications, as well.

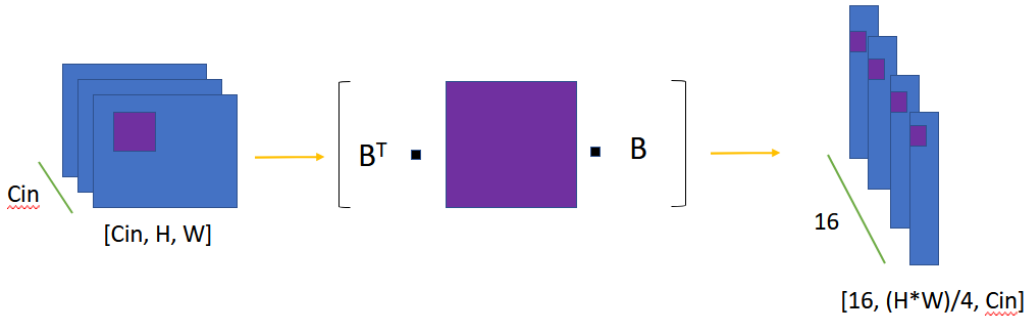


Figure 2.8: Transformation of an image with size (C_{in}, H, W).

For our architectures, we implemented $F((2 \times 2), (3 \times 3))$ minimal filtering algorithm. As mentioned in Chapter 2, we need $(2 + 3 - 1)^2 = 16$ input data to execute, $F((2 \times 2), (3 \times 3))$ algorithm. So, input images are divided into 4×4 image tiles for each channel in which, tiles are overlapped by 50%. Then tiles are tranformed by multiplying them with B^T and B matrices. Transformed values are stored in 3D matrices with size, $(16, \frac{H*W}{4}, C_{in})$ as given in Figure 2.8.

Similary, kernels of each filter are tranformed by multipliying them with G^T and G and transformed values are stored in 3D matrices with size, $(16, C_{out}, C_{in})$ as given in Figure 2.9. While training, filters are transformed for every iterations however for validation, they are transformed only once.

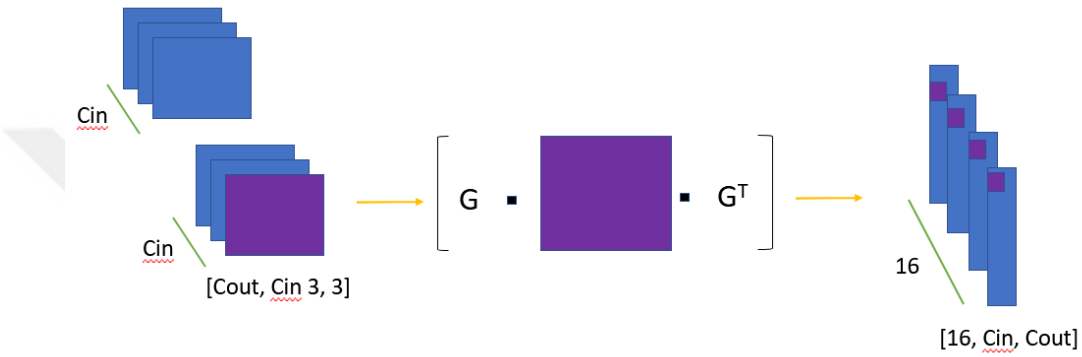


Figure 2.9: Transformation of filters with size $(C_{out}, C_{in}, 3, 3)$.

Transformed inputs and filters which are stored in 16 channels, are multiplied with parallel 16 GEMM operations, which is called batched-GEMM. In each GEMM, $(\frac{H*W}{4}, C_{in})$ sized matrices are multiplied with (C_{in}, C_{out}) sized matrices. Resulting outputs, $(\frac{H*W}{4}, C_{out})$ are stored in 16 channels. For batched-GEMM, the benchmark provides an optimized built-in implementation so we could implement 16 GEMM operations without needing any for loop.

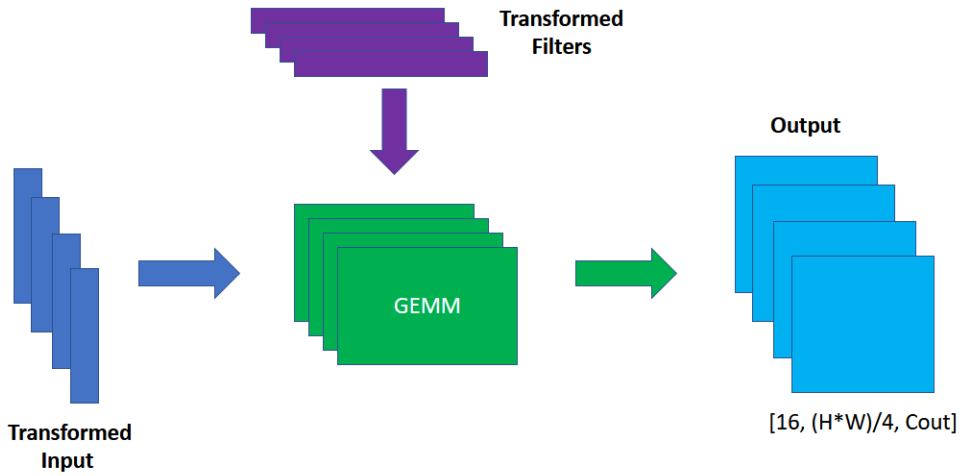


Figure 2.10: Batched GEMM for transformed input and filter multiplications.

Finally, resulting outputs across channels combined to form 4×4 tiles, then each tile multiplied with A^T and A matrices as shown in Figure 2.11. Resulting inversely tranformed outputs with size, 2×2 are stored in spatial domain with size (C_{out}, H, W).

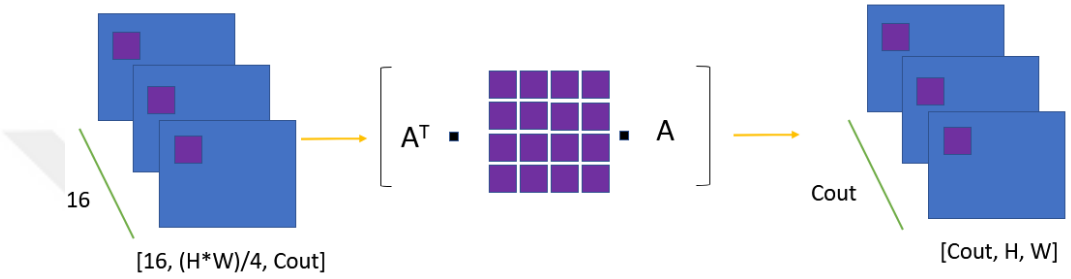


Figure 2.11: Inverse transform for resulting output of tranformed input and filter combinations.

2.3.2 Backpropagation with F(3x3,2x2)

Gradients are computed with respect to input and filter values while training a network. For a convolutional layer, gradient of inputs with respect to loss function is convolution of flipped filters with backpropagated error[2]. So if we think next layer's backpropagated error as an input image with size $H \times W$, we can use $F(2 \times 2, 3 \times 3)$ to compute gradients.

Gradients of weights however can be computed as the convolution of input values with next layer backpropogated error [2]. In other words, we are trying to compute $r \times r$ output with an $H \times W$ filter which can be denoted as $F(r \times r, H \times W)$. This is inapplicable since $H \times W$ filter size is not minimal for $F(m \times m, r \times r)$ algorithm. For this problem, [2] suggests that for a kernel size of 3x3 and 2x2 output, convolution can be decomposed into sum of smaller convolutions. So, we can compute gradients with respect to weights with $F(3 \times 3, 2 \times 2)$. To make it clear, backpropagated error is divided into 2x2 error tiles and then input tiles which are used for forward propagation is filtered with these 2x2 error tiles. So rather than 3x3 kernel we use 2x2 kernels to filter input values. Then, to form

$F(3 \times 3, H \times W)$, results of $F(3 \times 3, 2 \times 2)$ which is given in Figure 2.12 are accumulated over all tiles.

As you notice transformation matrices, A^T, G, B^T, are different from $F(2 \times 2, 3 \times 3)$, the transformation matrices for $F(3 \times 3, 2 \times 2)$ are given below,

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$G = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0.5 & -0.5 \\ 0 & 1 \end{bmatrix} \tag{2.13}$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Again using the equation given in (2.10), number of multiplications can be computed as $(3 + 2 - 1)(3 + 2 - 1) = 16$ which provides 2.25 arithmetic complexity reduction as we measure in forward propagation with $F(2 \times 2, 3 \times 3)$.

2.3.3 Arithmetic Complexity Analysis

Arithmetic complexity of a model can be measured with respect to number of multiplications or floating point operations [38]. For a 2D minimal filtering algorithm, $F(m \times m, r \times r)$, number of multiplications in convolutional layers is:

$$M = N[H/m][W/m]C_{in}C_{out}(m + r - 1)(m + r - 1) \tag{2.14}$$

Here N is the batch size, C_{in} is the number of input channels and C_{out} is the number of output channels. Batch is subset of images which are feeded to network every epochs. For a standard convolutional layer, number of multiplications is

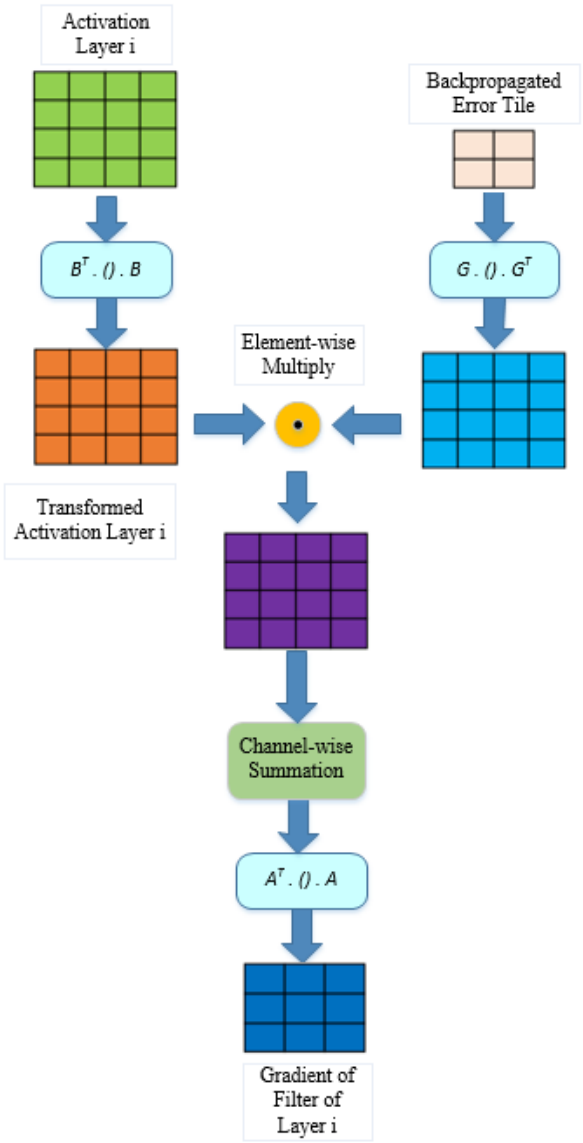


Figure 2.12: Winograd algorithm for $F(3 \times 3, 2 \times 2)$ while computing gradients of weights.

equal to $F(m \times m, r \times r)$ when $m = 1$.

Formula given in (2.14) can be simplified as:

$$\begin{aligned} M &= (m + r - 1)^2 / m^2 NHWC_{in}C_{out} \\ &= \alpha' NHWC_{in}C_{out} \end{aligned} \tag{2.15}$$

where $\alpha = (m + r - 1)^2$ and $\alpha' = \alpha / m^2$.

As it is mention in earlier parts, Winograd minimal filtering algorithm has additional addition operations to transform inputs, filters and outputs. Arithmetic complexity of the algorithm which can be defined as number of multiplications and addition operations performed, increases with these additional transformation. Arithmetic complexity for each transformation is written as:

$$\begin{aligned} T(D) &= \beta / m^2 NHWC_{in} \\ T(F) &= \gamma C_{in}C_{out} \\ T(I) &= \delta / m^2 NHWC_{out} \end{aligned} \tag{2.16}$$

Here, $T(D)$, $T(F)$ and $T(I)$ denotes arithmetic complexity of input transformation, filter transformation and inverse transform respectively and β, γ, δ values are the number of floating point instructions used by each transformation over small tiles [2].

Since direct convolution has no transformation, it would be more appropriate to measure arithmetic complexity of transformations relative to number of multiplications. Therefore, we can rewrite arithmetic complexity of transformations by dividing them with number of multiplications, M:

$$\begin{aligned} T(D)/M &= \beta / (C_{out}\alpha^2) = \beta' / C_{out} \\ T(F)/M &= \gamma / (NHW\alpha^2 / m^2) \\ &= \gamma / (P\alpha^2) = \gamma' / P \\ T(I)/M &= \delta / (C_{in}\alpha^2) = \delta' / C_{in} \end{aligned} \tag{2.17}$$

β', γ' and δ' are called as normalized arithmetic complexities of input, filter and

inverse transform, respectively [2]. Their values for different tile sizes are given in Table 2.1. Therefore total arithmetic complexity of $F(m \times m, r \times r)$ can be computed by adding each terms.

$$L = \alpha'(1 + \beta'/C_{out} + \gamma'/P + \delta'/C_{in})NHWC_{in}C_{out} \tag{2.18}$$

As we can notice, to reduce arithmetic complexity and energy consumption it is necessary to have small multiplication complexity, α' and smaller β', γ' and δ' values compared to C_{out}, P and C_{in} respectively.

Table 2.1: *Normalized arithmetic complexities multiplication(α'), input transform(β'), filter transform(γ') and inverse transform(δ') for image tile sizes.Tile size of $F(2 \times 2, 3 \times 3)$ is 4 [2].*

Tile	α'	β'	γ'	δ'
3	9.00	6.50	2.23	4.38
4	**4.00**	**2.00**	**1.75**	**1.50**
5	2.78	3.60	2.24	2.24
6	2.25	4.33	2.00	2.78
8	1.78	6.50	2.23	4.38

For a standard convolutional layer arithmetic complexity can be computed using (2.18) when $\alpha' = r^2$ and β', γ' and δ' values are zero. Thus, maximum arithmetic complexity reduction can be computed as R^2/γ^α which is 2.25 for $F(2 \times 2, 3 \times 3)$.

Chapter 3

Low Precision Arithmetic

3.1 Introduction

In this chapter, binarization techniques and gradient computations proposed in [18] are examined in detail. Furthermore, a linear quantization scheme is given to quantize paramaters of a network more than 1-bit.

3.2 Binarized Neural Nets

Binarized Neural Net, BNN, restricts real valued weights and activations used in both forward and backward phase to be binary. They are constrained to only two values, -1 or 1. So rather than using 32-bit or 64-bit data in arithmetic computations, 1-bit data is used which reduces memory size required for model elements and also ease memory access.

3.2.1 Deterministic and Stochastic Binarization

In [21] it is shown that constraining real valued parameters to -1 or 1, provides advantegeous for hardware implementation. So, real-valued weights or activations are restricted to -1 or 1 using deterministic and stochastic functions are given in [18].

Deterministic function is very straightforward and easy to implement,

$$w^b = \text{Sign}(w) = \begin{cases} +1 & \text{if } w \geq 0 \\ -1 & \text{if otherwise} \end{cases} \tag{3.1}$$

here w^b is binarized value.

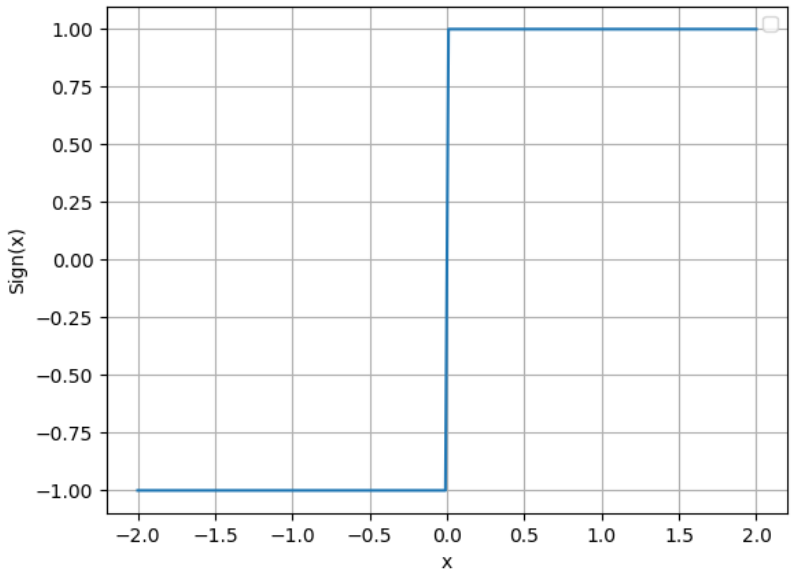


Figure 3.1: Deterministic binarization with Sign function.

Stochastic function however constraints values either -1 or 1 with a certain probability,

$$w^b = \text{Sign}(w) = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{3.2}$$

where σ is $hard - sigmoid$ function which is formulated as,

$$\sigma(w) = \text{clip}(\frac{w+1}{2}, 0, 1) = \max(0, \min(1, \frac{w+1}{2})) \tag{3.3}$$

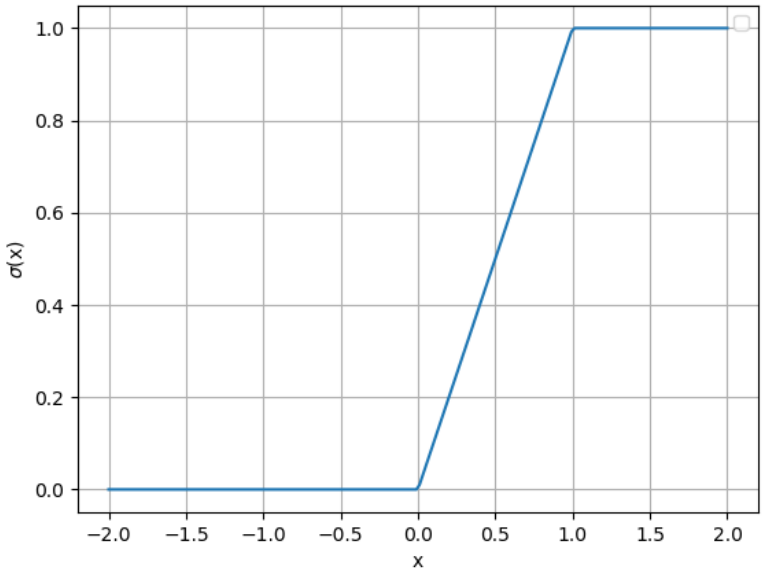


Figure 3.2: Hard sigmoid function for stochastic binarization.

Although stochastic binarization seems more engaging, it requires random bits to generate when binarize. So, for binarization, using deterministic functions like the sign function is more practical [21].

3.2.2 Gradient Computation and Propagating Gradients

Gradient-based optimization algorithms, like stochastic gradient descent, SGD, and Adam [39] search parameter space with infinitesimal and noisy steps. So, they require high precision gradients to reach local minima. Because of that, although binary weights are used to compute gradients of parameters, real-valued gradients are accumulated to update parameters.

Derivative of discrete functions like sign function is zero almost everywhere,

which is impractical for backpropagation algorithm. So, rather than computing gradients exactly, an estimate of them might be used. Bengio, et al. [40] is stated that most effective computations is achieved by state through estimator, STE, which is introduced in Coursera lectures of Hinton [41].

Consider deterministic binarization,

$$b = \text{Sign}(w) \tag{3.4}$$

and assume g_b is an estimator for gradient $\partial L/\partial b$ and it is obtained. Then, STE of gradient $\partial L/\partial w$ can be written as,

$$g_w = g_b 1_{|w|\leq 1} \tag{3.5}$$

where $1_{w\leq 1}$ is an estimator whose value is 1 if given condition, $|w| \leq 1$ is satisfied else it is zero. As we notice for large magnitude of w, gradient of it is zero. It constraints gradients to be in between [-1, 1]. This prevents weights to overflow with exploded gradients. When it is not zeroed, network performance might be drastically reduced.

3.2.3 Binarization as Requlazier

Binarizing weights can be seen as a requlariza tion since most regularization techniques [42], [43], [44], [45] achieve better generalization by adding noise to weights and activations. For example, Dropout [44] sets half of the weights to zero randomly while measuring gradients of them. It can be defined as adding noise to certain elements which make them zero.So, from this point of view, it can be stated that binarization defined in Section 3.2.1, is an alternative of Dropout.

3.3 Linear Quantization

For tasks which are less tolerant to accuracy loss, like pedestrian detection, Binarized Neural Networks, could not achieve desired level of accuracies. So, for these task, rather than binarizing parameters, we can quantize them to more than 1-bit. In, [46] so called linear quantization scheme is suggested to quantize weights. Basically, it converts 32-bit floating point values to low precision fixed-point values.For a given input x and the bitwidth, the linear quantization operation is defined as:

$$\textbf{LinearQuant}(x, bitwidth) = \text{Clip}\Big(\text{round}\Big(\frac{x}{2^{bitwidth}-1}\Big)\times 2^{bitwidth-1}, \text{minV}, \text{maxV}\Big) \tag{3.6}$$

Here, minV and maxV are the minimum and maximum scale ranges which can be measured as,

$$\text{maxV} = 2^{bitwidth-1}, \quad \text{minV} = -2^{bitwidth-1} \tag{3.7}$$

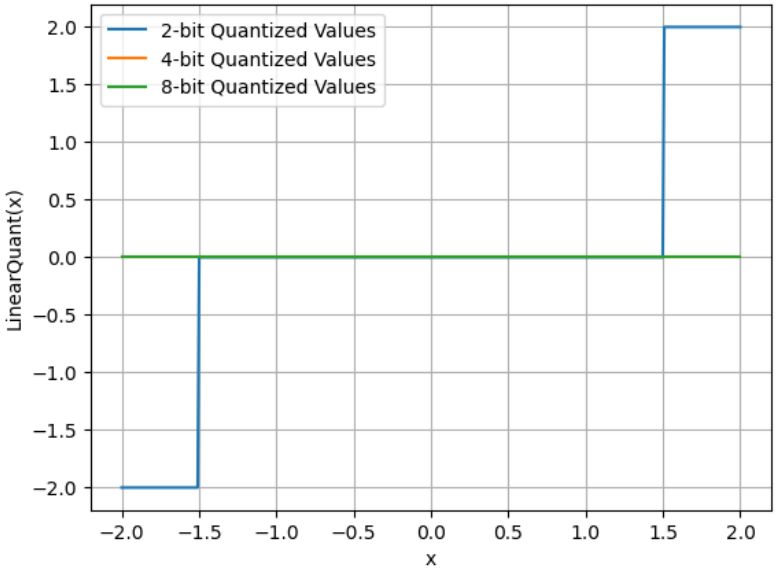


Figure 3.3: 32-bit values between [-2, 2] are quantized to 2-bit, 4-bit and 8-bit with given Linear Quantization scheme in 3.6.

In state of the art CNN architectures, weights and activations are around zero mean or normalized to have zero mean. So, in our experiences we observed that quantizing weights or activations with this scheme results zero terms instead of quantized values. In Figure 3.3, 32-bit floating point values between -2 and 2 are quantized to 2-bit, 4-bit and 8-bit with this linear quantization scheme and results are plotted. The reason for this is division operation acts as left shift of radix point which makes integer parts zero so after rounding we get zero values. Therefore, we propose another quantization scheme by changing division operation with multiplication which acts as right shift of radix point. Our proposed scheme is formulated as,

$$\textbf{LinearQuant}(x, bitwidth) = \text{Clip}\Big(\Big(\frac{\text{round}(x \times 2^{bitwidth-1})}{2^{bitwidth} - 1}\Big), \text{minV}, \text{maxV}\Big) \tag{3.8}$$

Here maxV and minV values can be measured as given in 3.7.

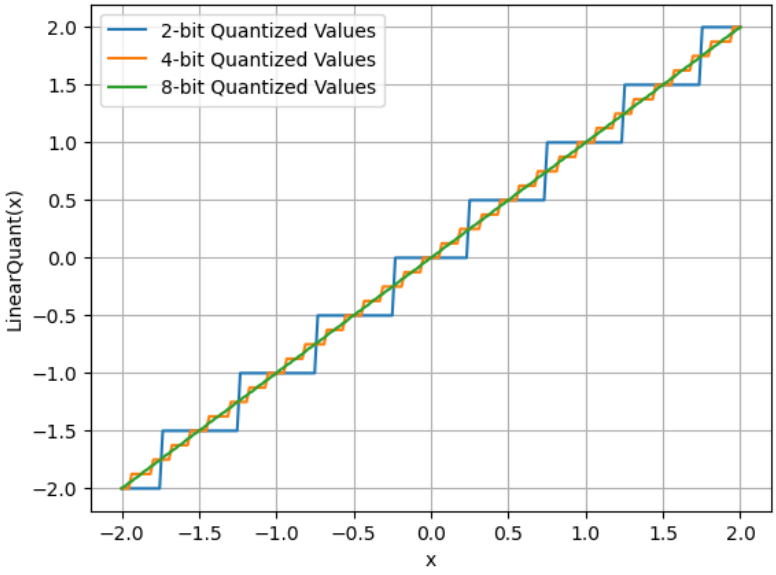


Figure 3.4: 32-bit values between [-2, 2] are quantized to 2-bit, 4-bit and 8-bit with proposed Linear Quantization scheme in 3.8.

Our proposed quantization scheme whose resulting outputs for 32-bit floating point values between [-2, 2] given in Figure 3.4, achieves desired accuracy levels on CIFAR10 and CIFAR100 dataset. So, emprically it can be stated that proposed

quantization scheme (3.8) is much more preferable than (3.6).

3.4 Power Efficiency and Memory Access

In [3], Horowitz provides rough power consumption measurements for multiply-accumulate, MAC, operations and memory accesses on 45 nm process nodes as given in Table 3.1 and 3.2. By looking at the tables, we can observe that memory access requires more power than MAC operations so by reducing memory access, energy required for training or inference might be reduced.

Quantization of weights reduce memory size required to store weights and memory access significant amount. For binary weights, for example, memory size required to store weights is just 1-bit. When we think 32-bit float weights used in common benchmarks like Torch and Tensorflow, this means x32 smaller memory space and x32 less memory access. So we can state that quantization drastically reduces the energy needed for training and also inference.

Table 3.1: *Power consumption of MAC operations which are performed on 45 nm process nodes [3].*

Operation	**MUL**	**ADD**
8bit Integer	0.2pJ	0.03pJ
32bit Integer	3pJ	0.1pJ
16bit Floating point	1pJ	0.4pJ
32bit Floating point	4pJ	0.9pJ

Table 3.2: *Power consumption for memory access given by RAM memory sizes on 45nm process node [3].*

Memory size	**64-bit memory access**
8KB	10pJ
12KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

Chapter 4

Dynamic Sparsity of Activations

Sparsification, also called pruning can be simply defined as removing neurons and weights from the network. It can be imposed after training a fully-sized network or during training phase. Ideally, for a given performance task, weights or units of a network which provide least benefit to task can be pruned. Pruning a network after fully trained, gains benefits just in inference time. However, with dynamic sparsification, in which we prune weights or units during training time, we can benefit from sparsity in training phase.

Deep CNNs have many neurons with very low activations regardless of data presented [47]. These neurons are likely to be redundant and have no effect on performance. So, they can be dropped out without effecting overall accuracy. To exclude these neurons, in our Winograd CNN architectures, we use ReLU activation function and Targeted Dropout [29]. So, we built sparse Winograd CNN architectures.

4.1 ReLU for Sparse Neurons

ReLU function rectifies the activations of a network whose values are negative, to zero which allows the network to learn non-linear relations between parameters.

Besides, it gives sparse representation of the activations which provides aggressive pruning in traning phase.

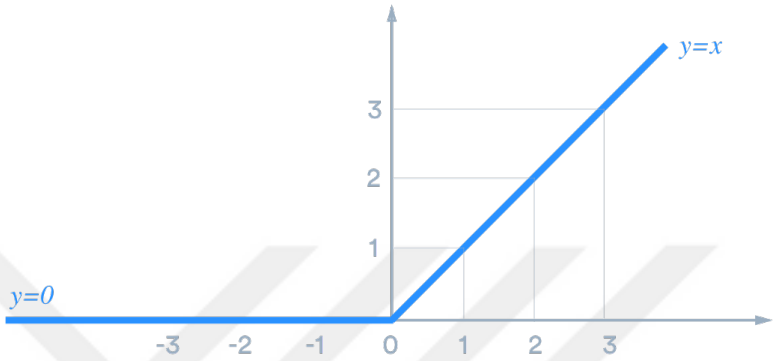


Figure 4.1: ReLU namely Rectified Linear Unit Activation Function, $y = \max(0, x)$.

4.2 Targeted Dropout

Dropout [43], is a widely used technique for regularization of large scale neural networks. It randomly omits activation units from the network for each training step, which prevents feature detector to be correlated with specific feature detectors. In practise, Dropout generates a mask whose values are sampled from, $M \sim \text{Bernoulli}(1-\beta)$ and masks the activation or input units with this mask. Bernoulli$(1-\beta)$ generates random variables whose values 1 with probability $p = (1-\beta)$, otherwise 0. For a convolutional layer with an input tile X, filter matrix W and output matrix Z, Dropout can be defined as,

$$Z = (X \odot M) \times W \tag{4.1}$$

Dropconnect [45] is a variant of Dropout, that omits subset of weights randomly from the network rather than activation units for each training iteration. It forces the network to learn alternative connections. In practise, it also generates a mask whose values are sampled from, $M \sim \text{Bernoulli}(1-\beta)$ and masks the weights with this mask. Dropconnect can be formulated as,

$$Z = X \times (W \odot M) \tag{4.2}$$

Targeted Dropout, [29], is another method for dropping activation units and weights of the network. Contrary to Dropout and Dropconnect, it stochastically selects subset of weights and units to be dropped by using simple sparsity criterion and then randomly drops activations or weights among these subsets for each training step. The method uses simple but easy to implement magnitude based sparsity criterion rather than more complicated sparsifying methods.

Magnitude-based sparsifying, so called magnitude based pruning techniques are commonly used pruning techniques. Those techniques treat top-k parameters with largest magnitude as sufficient to train network, so remaining parameters can be omitted. Parameters are selected with respect to their L_1 and L_2 norms. Depending on pruning startegy, whether you prune unit or weight, other than the top-k values which maximize L_1 or L_2 norms of the weight matrix selected as candidate for pruning.

For unit pruning, L_2 norm of column vector of weights are measured for each corresponding filter,

$$W(\theta) = \left\{ \text{argmax}_{w_o} \|w_o\|, 1 \leq o \leq N_{col}(w) \right\} \tag{4.3}$$

here w_o is column vector of filters with size $C_{in} \times r \times r$ and $N_{col}(w)$ is the number of output channel, C_{out} of that convolutional layer. To make it clear, filters of the convolutional layer, are converted to column vector, and then their L_2 norms are calculated. Filters with lowest L_2 norm are selected to be dropped, and finally the filters are omitted, randomly.

For weight pruning, weights in each filter are sorted with respect to their L_1 norms and the ones with lowest magnitude are selected as candidate for dropping.

$$W(\theta) = \left\{ \text{argmax}_{W_{io}} |W_{io}|, 1 \leq i \leq N_{row}(w) \right\} \tag{4.4}$$

Here, $N_{row}(w)$ is the number of elements in a filter.

Although, weight pruning is studied commonly in large scale for deep neural networks. It provides worthwhile computational efficiency under adequately sparse states and requires sparse linear algebra equations to implement. However, unit pruning can be implemented with standard linear algebra equations, which might provide much computational efficiency for a fixed sparsity level [29].

Number of parameters, to dropout is selected with a stochastic process. First bottom $\delta|\theta|$ values are selected as candidate parameters to be dropped out where δ is target proportion, θ is the parameter space and $|\theta|$ is number of parameters, and then among these parameters, some variables are dropped with a drop rate, β. So, the total number of parameters to be dropped for each iteration can be calculated as $\delta \cdot \beta|\theta|$ [29].

An example of Targeted Dropout for unit pruning is shown in Figure 4.2. Here, a filter with lowest L_2 norm is pruned, then after multiplication corresponding output, which is the input of next layer is also pruned. So, we sparse the units of next convolutional layer which will reduce the number of floating operations substantially.

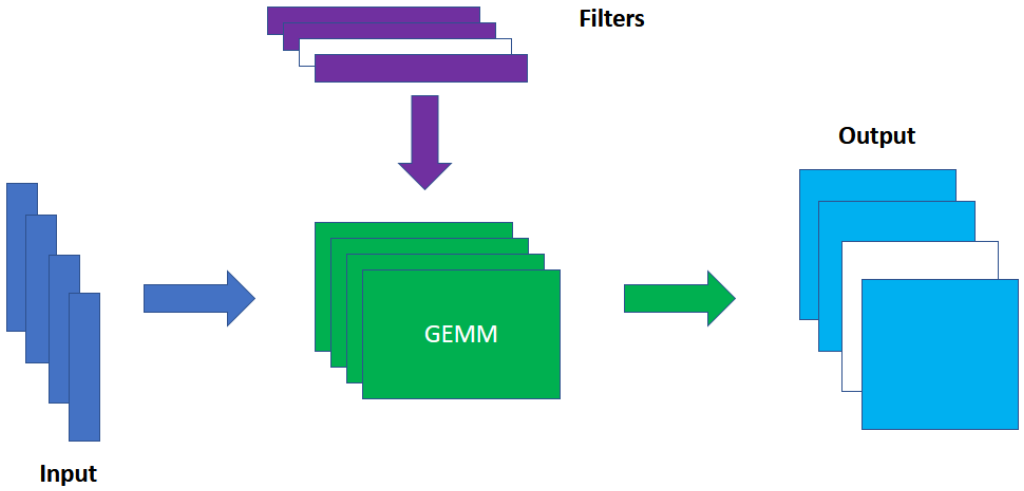


Figure 4.2: Unit pruning example with Targeted Dropout.

Chapter 5

Experiments and Results

We conducted several experiments to examine the effect of algebraic property of convolution, low precision arithmetics and dynamic sparsity of activations on classification performance of deep convolutional neural networks. Conducted experiments are listed as follows,

- Convolutional layers for CNN models are implemented with $F(2 \times 2, 3 \times 3)$ minimal filtering algorithm,
- ReLU activation layer moved after input tranformation and filter transformation excluded from conventional Winograd algorithm rather 4x4 kernels initialized. We call this algorithm as ReLU-ed Winograd algorithm,
- Weights of the ReLU-ed Winograd algorithm are quantized to be binary, 2-bit, 4-bit and 8-bit,
- ReLU operation is completely moved from the network, rather activation units are pruned for different target proportion, δ and drop rate, β values.

Each experiment is executed by implementations based on Torch [48] which is a commonly used benchmark for machine learning and deep learning tasks. Network architectures given in Table 5.1 and 5.2 are trained and validated over CIFAR10 and CIFAR100 datasets.

Table 5.1: *Network architecture used to train and inference conventional Winograd CNN model on CIFAR10 dataset. For CIFAR100, bottom linear layer's output unit changes to be 100.*

Layer	**Filter Size**
Conv	(128, 3, 3, 3)
BatchNorm2d	-
ReLU	-
Conv	(128, 128, 3, 3)
MaxPool2d	-
BatchNorm2d	-
ReLU	-
Conv	(256, 128, 3, 3)
BatchNorm2d	-
ReLU	-
Conv	(256, 256, 3, 3)
MaxPool2d	-
BatchNorm2d	-
ReLU	-
Conv	(512, 256, 3, 3)
BatchNorm2d	-
ReLU	-
Conv	(512, 512, 3, 3)
MaxPool2d	-
BatchNorm2d	-
ReLU	-
Linear	(8192, 1024)
BatchNorm1d	-
ReLU	-
Linear	(1024, 1024)
BatchNorm1d	-
ReLU	-
Linear	(1024, 10)
BatchNorm1d	-
LogSoftmax	-

Table 5.2: *Network architecture used to train and inference ReLU-ed Winograd CNN, Binary ReLu-ed Winograd CNN as well as Quantized networks and Sparse Winograd CNN models on CIFAR10 dataset. For CIFAR100, bottom linear layer's output unit changes to be 100.*

Layer	**Filter Size**
Conv	(128, 3, 4, 4)
BatchNorm2d	-
Conv	(128, 128, 4, 4)
MaxPool2d	-
BatchNorm2d	-
Conv	(256, 128, 4, 4)
BatchNorm2d	-
Conv	(256, 256, 4, 4)
MaxPool2d	-
BatchNorm2d	-
Conv	(512, 256, 4, 4)
BatchNorm2d	-
Conv	(512, 512, 4, 4)
MaxPool2d	-
BatchNorm2d	-
Linear	(8192, 1024)
BatchNorm1d	-
ReLU	-
Linear	(1024, 1024)
BatchNorm1d	-
ReLU	-
Linear	(1024, 10)
BatchNorm1d	-
LogSoftmax	-

5.1 Preprocessing and Data Augmentation

CIFAR10 is an image classification dataset, which consists of a training set of 50K images and a test set of 10K images. The images are 32x32 colorful images. They represent 10 labels which are, automobiles, airplanes, birds, cats, dogs, deer, frogs, horses, ships and trucks. CIFAR100 just like CIFAR10, contains 50K training images and 10K images, but it has 100 labels so number of images per label is 600. For small datasets like CIFAR10 and CIFAR100 preprocessing and data augmention can affect classification accuracy significantly. Therefore, we use preprocessing and data augmentation techniques defined in literature before training.

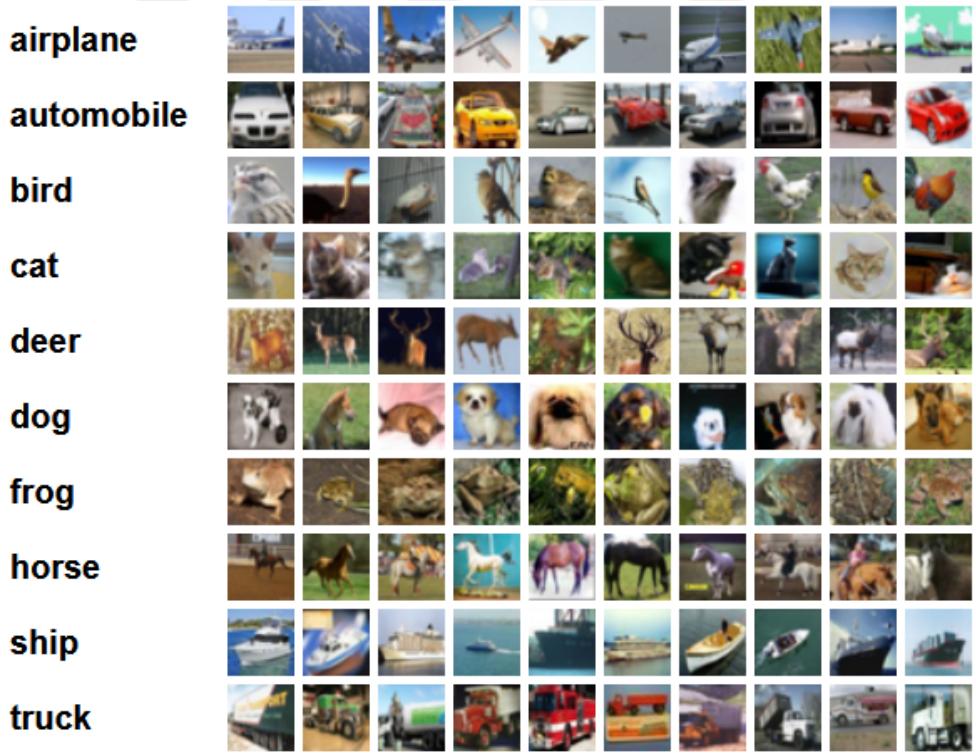


Figure 5.1: Sample images from CIFAR10 dataset [1].

To preprocess data, we use z-score normalization. The motivation behind the normalization is to ensure that all feature variables are in the same scale. Because, contribution of variables differs for different scales and learning can be biased to

specific variables. Basically, for z-score normalization, also called standardization, channel-wise means are subtracted from image pixels and then resulting normalized pixels are divided by channel-wise standard deviations. Channel-wise mean, μ and standart deviations, σ for CIFAR10 dataset are estimated as,

$$\mu = [0.485, 0.456, 0.406]$$
$$\sigma = [0.229, 0.224, 0.225]$$

For CIFAR100, these values are estimated as,

$$\mu = [0.507, 0.486, 0.441]$$
$$\sigma = [0.267, 0.256, 0.276]$$

Data augmentation can be defined as adding synthetic data, which is generated by data tranformations to dataset. It provides us to expose additional variations of data to network without any effort to collect and label data. Since CNNs are rotation invariant, they should also learn mirror of an image as feature detector. So, we select random images from the whole set and flip these images horizontally, which is called random horizontal flipping. Moreover, we take random snapshots from image and rescaled them to original sizes of the image. This operation is called random cropping. One benefit of cropping is that it reduces background effect on feature detectors. Second benefit is that, in certain images labeled object might not be recognized easily. In other words, if the labeled object is in a small area of the picture, detector can not extract its features effectively. By cropping, we can make these regions more visible. One tricky thing about cropping is that cropped region cannot be wholly on the image. So to solve this, padding is needed. We use padding as 4 for our croppings.

5.2 Modified Winograd Arhitectures

To experience the effect of low precision arithmetics and activation sparsity, conventional Winograd algorithm given in Figure 5.2a is modified.

5.2.1 ReLU-ed Winograd CNN

As we mentioned in Chapter 4, ReLU function provides dynamic sparisty for activations. However, for Winograd minimal filtering algorithms, the use of ReLU activation function just provides a network to learn non-linear relations but not sparsity. Because, the activations of previous layer are transformed before multiplied with the filters. So, in order to benefit from sparsity, [49] suggest to move ReLU operation after input transformation. Although it changes Winograd architecture, it reduces computational complexity, substantially.

The design of conventional Winograd algorithm and modified version of it, are given in Figure 5.2b. As we observe, for ReLU-ed Winograd, besides moving ReLU, transformation of filter in transform domain is excluded from the architecture. Rather we initialize filters with 4x4 kernels and use them for both training and inference. This provides us to get rid of filter transformation cost. So, we can rewrite total arithmetic complexity of $F(m \times m, r \times r)$ given in 2.15 by moving γ'/P term out as,

$$L = \alpha'(1 + \beta'/K + \delta'/C)NHWCK \tag{5.1}$$

After these modifications, resulting output of 2D minimal filtering algorithm defined in 2.9, can be redefined as,

$$Y = A^T[W \odot \text{ReLU}(B^T X B)]A \tag{5.2}$$

where W is a 4x4 kernel.

Although in most CNN architectures kernel sizes are odd, there is no restriction to be them even. By changing kernel size, we benefit from computational

complexity but in return power needed to store and access to kernels are increased. As we mentioned in earlier chapters, memory access can be very costly so to reduce memory access needed, weights of the network are quantized.

5.2.2 Quantization of Weights

Linear Quantization scheme proposed in (3.8) is applied to get 2-bit, 4-bit and 8-bit representations of 32-bit floating point weights and deterministic binarization defined in (3.1) is used to binarize weights. After binarization ReLU-ed Winograd algorithms is changed as given in Figure 5.3. We name this algorithm as Binary ReLU-ed Winograd algorithm. New algorithm can be formulated as,

$$Y = A^T[(\text{Binarize}(W) \odot \text{ReLU}(B^T X B)]A \tag{5.3}$$

For quantized network we use the same architecture, just Binarize($\odot$) operation changed by Quantize($\odot$) operation.

While training quantized networks, weight initilization gains more importance. Since if the weights are quantized a bitwidth where radix point, known as decimal point, is shifted to a point for which integer part is zero, quantized values becomes zero after rounding. To tackle with this problem, weights which are initalized with Xavier Uniform initializer [50] are scaled by a gain.

Xavier Uniform initializer basically, drawns samples from a uniform distribution, U(-m,m) where m is defined as,

$$m = gain \times \frac{\sqrt{6}}{\sqrt{c_{in} + c_{out}}} \tag{5.4}$$

Depending on number of input and output channels in the network, scaling values changes. For the network architecture given in Table 5.2, initalized real valued weights are scaled by 16, 4 and 1 for 2-bit, 4-bit and 8-bit quantizations respectively.

5.2.3 Pruning Redundant Inputs from Network

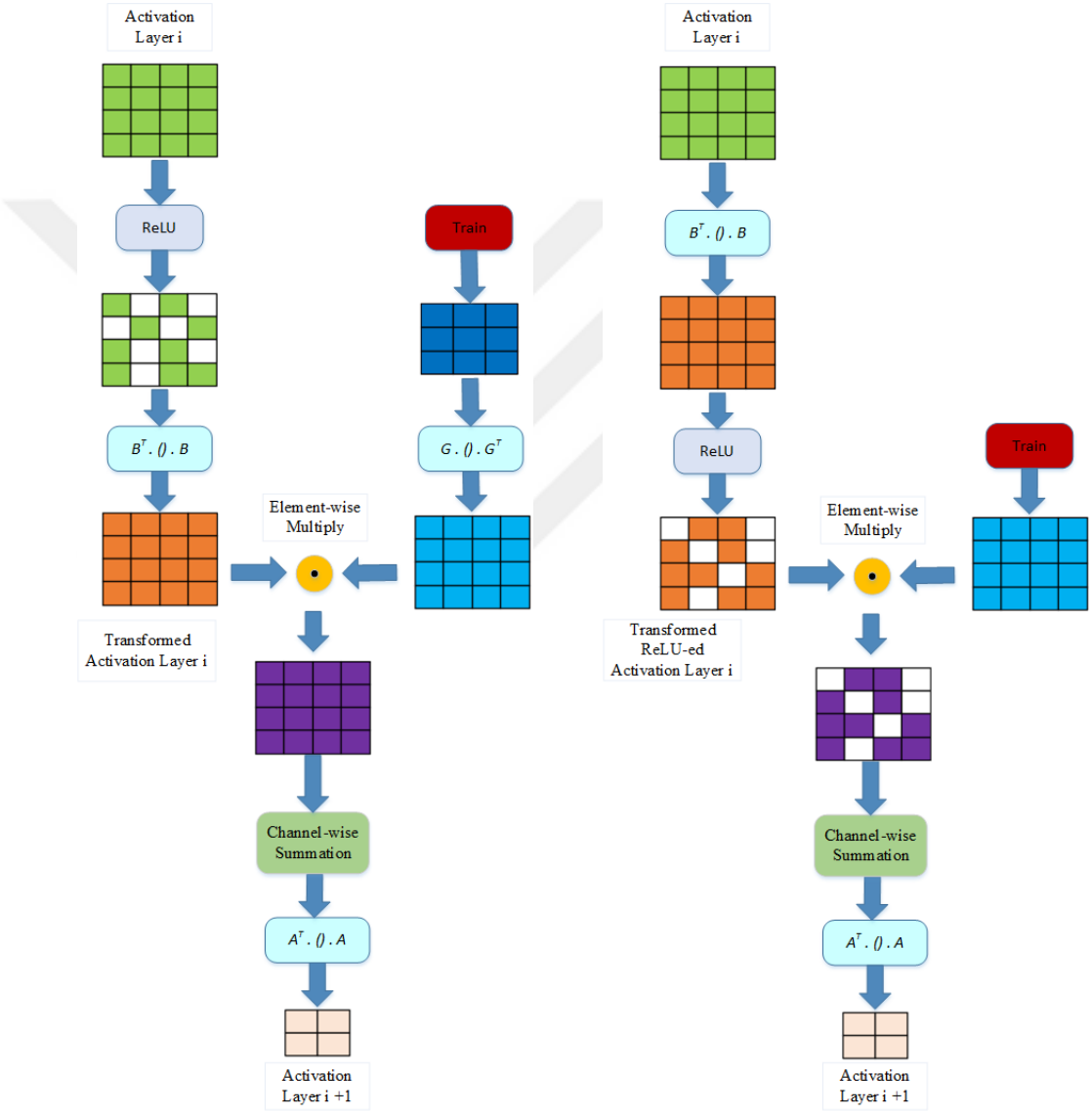
As we discussed in earlier chapters, modern CNN architectures have redundant data that have no benefit for classification task so they can be pruned while training models. For conventional Winograd algorithm rather than moving ReLU operation we apply Targeted Dropout to prune redundant input values from the network. It provides us to control sparsity rate of convolutional layers besides computational efficiency.

Unit pruning in convolutional layers is actually pruning filters, which can be called group sparsity. As we discussed in Chapter 4, for each filter, weights are converted to column vectors and L_2 norms of each filter are measured over these column vectors. The ones with lowest L_2 norm are selected to be dropped and among them, filters are dropped randomly.

We train our networks, for different values of target proportion δ, and drop rate β. For each convolution layer we use same δ and β values. For our first model we use $\delta = 0.5$ and $\beta = 0.5$ so that around %25 of units are pruned. For our second model, we use $\delta = 0.75$ and $\beta = 0.66$ so that half of the units are pruned. We use architecture given Table 5.2 for our models and deploy no activation function to neurons.

5.3 Training Settings

For each model, the cross entropy loss is minimized with ADAM [39]. Learning rate α is initally set to 1e-4 and 1e-3 while training models given in Table 5.1 and 5.2 respectively.It is decayed in every 2 epochs, by multiplying it with 0.98. In other words, in every 2 epoch learning rate value changed to be as $0.98^{\text{epoch}//2} * \alpha$.



(a) Conventional Winograd algorithm (b) ReLU-ed Winograd algorithm

Figure 5.2: Combining Winograd convolution with sparse activations: (a) conventional Winograd-based algorithm for $F(2 \times 2, 3 \times 3)$ (b) ReLU-ed Winograd algorithm for $F(2 \times 2, 3 \times 3)$.

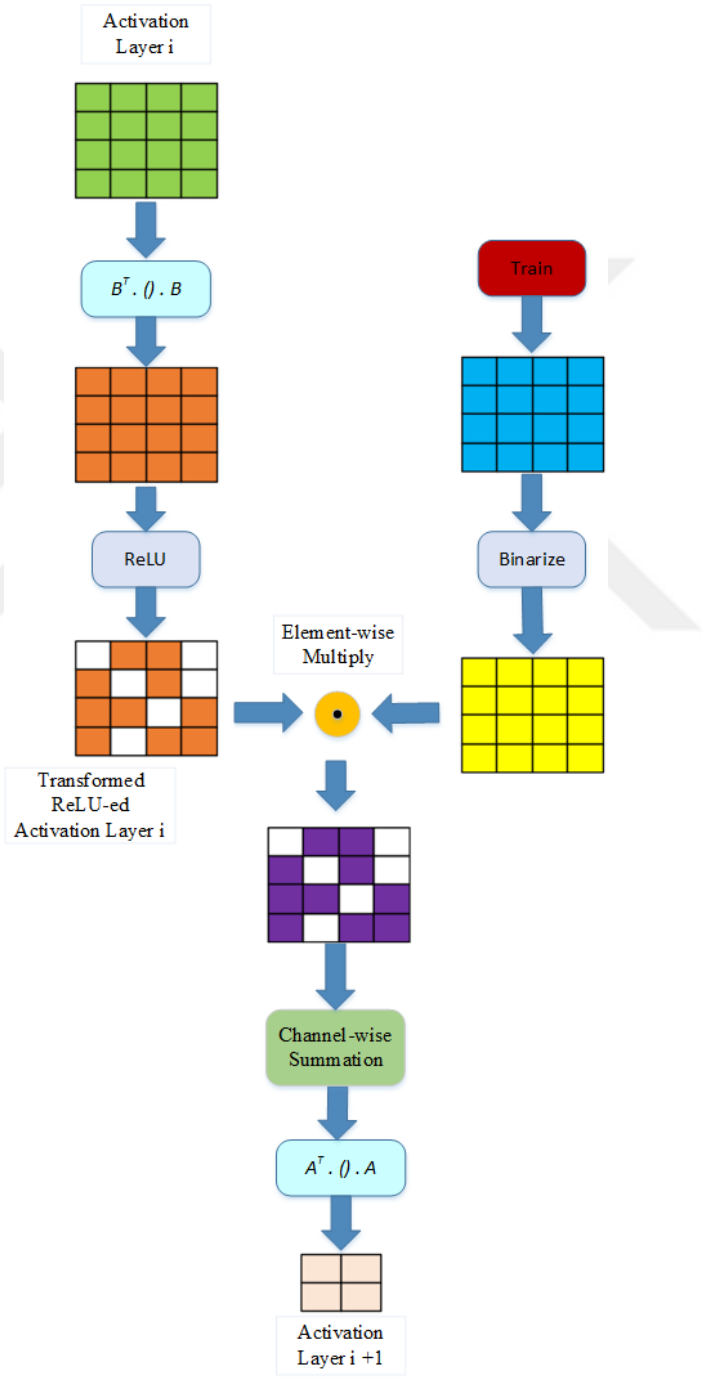


Figure 5.3: Binary ReLU-ed Winograd algorithm for $F(2 \times 2, 3 \times 3)$.

For multi-class classifications cross entropy loss can be defined as,

$$Loss(y, \widehat{y}) = -\sum_{k}^{K} y^k \log \widehat{y}^k \tag{5.5}$$

here $\widehat{y}^k$ takes values 0 or 1 which indicates whether class k classified correctly or not. y^k is ground-truth values for labels and K is the number of class in that dataset.

While training binarized weights, real-valued weights are clipped and bounded within [-1, 1] interval since, binarization is not affected by magnitude of parameters. And by clipping, parameter growing is limited to certain values. Actually bounding weights within certain intervals is a common practise. It can be used as regularization.

We use batch normalization to reduce internal covariate shift and achieve fast convergence which reduces number of training epochs. Batch normalization basically, standardizes mini-batches as we preprocess data. It scales and shifts the standardized data with scale parameter γ and shift parameter β.

$$y = \gamma\widehat{x} + \beta \tag{5.6}$$

where $\widehat{x}$ standardized value of x and y is the result of batch normalization. These parameters can be fixed or learned by training as well.

We use test sets of datasets as validation sets, so training is executed over whole training sets. Test accuracy achieved is associated with best validation accuracy achieved after 150 epochs training.

5.4 Results

Test accuracies achieved after training conventional Winograd CNN and other modified CNNs on CIFAR10 and CIFAR100 datasets are reported in Table 5.3.

Model trained using conventional Winograd algorithm given in Figure 5.2a is called Winograd CNN. Other models which use the algorithms given in Figure 5.2b and 5.3 are called, ReLU-ed Winograd CNN and Binary ReLU-ed CNN respectively. Sparse Winograd CNNs, omits ReLU operation in ReLU-ed Winograd algorithm and sparse activation units with Targeted Dropout.

Table 5.3: *Test results for models achieved over CIFAR10 and CIFAR100 datasets.*

Model	**CIFAR10**	**CIFAR100**
Vanilla CNN	93.95%	72.56%
Winograd CNN	93.86%	72.78%
ReLU-ed Winograd CNN	92.01%	69.20%
Binary ReLU-ed Winograd CNN	90.08%	66.65%
BinaryConnect [18]	90.10%	-
Q2-bit ReLU-ed Winograd CNN	91.83%	64.11%
Q4-bit ReLU-ed Winograd CNN	91.91%	66.34%
Q8-bit ReLU-ed Winograd CNN	91.95%	68.37%
Sparse Winograd CNN($\delta = 0.5$,$\beta = 0.5$)	90.06%	-
Sparse Winograd CNN($\delta = 0.75$,$\beta = 0.66$)	88.59%	-

Chapter 6

Discussion

As we mentioned throughout the thesis, memory space required to store a model and memory needed for performing floating point operations on run-time, are the main constraints of resource limited devices. To deploy deep CNNs on these devices, we need to reduce and ease memory accesses. Memory space needed to store a model depends on number of parameters and network architecture; number of layers, number of filters and kernel size [38]. Run-time memory also depends on number of parameters and the number of floating point operations. Most of the parameters are stored and most of the number of floating operations are performed on convolutional layers so we investigated several approach to make these layers more efficient.

Convolutional layers of our models are implemented with $F(2\times2, 3\times3)$ minimal filtering algorithm, which reduces number of multiplications 2.25 times without any accuracy loss. They achieve almost the same accuracy levels with the models with direct convolutional layers. So far, their success and efficiency are limited by low kernel sizes, for the networks with large filter sizes like 5x5 and 11x11, their efficient implementations is an open research area. For the networks with large size filters FFT based algorithms can be used. Although there are architectures with large size filters like GoogleNet and ResNet, there is a trend to use 3x3 even 1x1 filters. For example Mobilenets which are specially designed for embedded

devices, use 3x3 and 1x1 filters. Therefore, we applied certain modifications and processes to make these algorithms more efficient.

ReLU operation and Targeted Dropout provide us to built sparse Winograd architectures. In recent years, sparse network architectures became quite popular.We can achieve almost the same accuracy level by using less memory space and less floating point operations. ReLU-ed Winograd architecture changes the conventional Winograd algorithm to make use of ReLU operation as sparsity of activations. It prunes transformed inputs aggresively with just $< 1.9\%$ accuracy loss. Main disadvantage of this algorithm is kernel size, although we benefit from filter transformation cost, we enlarge the memory space needed to store parameters. Besides, the ReLU-ed Winograd algorithm can not be used to inference the models trained with 3x3 kernels. However, we can train models from stracth using this architecture.

Targeted Dropout is a variant of Dropout. It uses magnitude based pruning techniques to prune both activations and weights. Contrary to ReLU operation, we can control the sparsity level of convolutional layers by changing target proportion, δ and drop rate β values. Moreover, Targeted Dropout can be stated as structured pruning technique for activations since it prunes the whole units combined with redundant filters. Compared to unstuctured pruning techniques which prune paramaters regardless of their region structured pruning provides better classification performance [51]. Main disadvantage of Targeted Dropout against ReLU-ed Winograd is selection of drop paramaters, δ and β for which tuning is required. However with optimal parameter selection we can achieve similar results with ReLU-ed Winograd CNN. For $\delta = 0.5$ and $\beta = 0.5$ almost 25% of parameters are dropped but we achieve almost the same accuracy level with Binary ReLU-ed Winograd CNN on CIFAR10 dataset. Another challenge for target dropout and actually for other pruning techniques, is executing them on hardwares. They require additional operations like evaluations of weights, sorting, random dropping etc. These increases run-time memory usage. However, with an optimized implementation their usage might become more efficient. Compared to other pruning techniques, [47], [27] Targeted Dropout is easy to implement and robust for post hoc prunings.

As we discussed, ReLU-ed Winograd algorithm increases the memory space requirements. To tackle with this problem, we quantize weights to very low precisions. By binarizing weights we reduce memory space requirement x32 times and convert floating point operations to bit-wise operations in return we incur less than 2% accuracy loss. Our result also matches up with BinaryConnect [18] whose weights are also binarized and uses same network architecture, number of layers and filter size. Again by quantizing real-valued weights to 2-bit, 4-bit and 8-bit fixed point weights we reduce memory sizes significantly with infinitesimal small accuracy losses on CIFAR10. However, for CIFAR100 dataset as we lower the precision, accuracy loss increases. Accuracy loss for the model trained with binary weights is around 6% which cannot be tolerated by real-time applications like pedestrian detection, face recognition. This is main disadvantage of binary neural nets. Their practical implementations change with accuracy requirement of performance tasks. However, 8-bit quantized networks achieves similar results compared to full precision networks, as we expected. Moreover, both binarization and quantization in a way requlrize the weights by adding noise to them. Binarization used for training can be stated as a variant of Dropout. Rather than setting parameters to zero randomly, it sets them to be -1 or 1.

To sum up, Winograd's minimal filtering algorithms provide us to reduce number of multiplications performed in a convolutional layer without accuracy loss. And these algorithms can be more effective with sparse network architecture proposed in Figure 5.2. Given architecture achieves almost the same results on both CIFAR10 and CIFAR100. Pruning activations with Targeted Dropout is an alternative of ReLU activation functions for ReLU-ed Winograd architecture. It achieves different results for different set of parameters and by tuning these parameters we can increase network performance. Moreover quantization is a game changer for memory reduction, spesifically binarization reduces memory space required both for storage and on run-time drastically, besides it can be easily implemented on hardware devices with limited resources. Best architecture differs with respect to performance task given. For tasks which are less tolerant to accuracy loss, 8-bit quantized ReLU-ed Winograd CNN can be optimal choice whereas for other applications Binary ReLU-ed Winograd CNN might be prefered.

Chapter 7

Conclusion and Future Work

In this thesis, we examined several approaches to built memory friendly and efficient deep convolutional neural networks. We optimized the direct convolution algorithm performed in convolutional layers with Winograd's minimal filtering algorithms and propose certain modifications to make these algorithms more memory efficient. We used ReLU and Targeted Dropout to sparse transformed inputs, dynamically so we can train sparse networks where half of the activations are zero. Moreover, we quantized weights to very low precision values. We binarized weights using deterministic binarization and propose a linear quantization scheme to quantize weights to more than 1-bit.With binarization we reduced the memory space requirements x32 times. The algorithm itself and modified architectures were evaluated on CIFAR10 and CIFAR100 dataset. We achieved state of the art results both with conventional Winograd and ReLU-ed Winograd algorithms. Depending on performance task given we chose between Binary ReLU-ed Winograd and 8-bit Qunatized ReLU-ed Winograd as optimal algorithm for convolutional layers.

As we discussed, efficient implementation of Winograd' s minimal filtering algorithm with large size filters is an open research area . Also, Strassen algorithms [52] can be applied to decrease multiplication complexity of the minimal filtering algorithms. Moreover, rather than linear quantization, logarithmic data

representation scheme [46] can be used to increase classification performance of the networks with very low precision parameters. These points require further investigation.

Bibliography

[1] "Cifar10 dataset." `https://www.cs.toronto.edu/~kriz/cifar.html`, (accessed October 26, 2020).

[2] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[3] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[5] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[7] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2736–2744, 2017.

[8] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[9] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International conference on machine learning*, pp. 2285–2294, 2015.

[10] 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.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[15] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[16] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[17] 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.

[18] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, pp. 3123–3131, 2015.

[19] D. Soudry, I. Hubara, and R. Meir, "Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights," in *Advances in neural information processing systems*, pp. 963–971, 2014.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[23] Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization," in *International Conference on Machine Learning*, pp. 242–252, PMLR, 2019.

[24] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, pp. 598–605, 1990.

[25] B. Hassibi, D. G. Stork, and G. J. Wolff, "Optimal brain surgeon and general network pruning," in *IEEE international conference on neural networks*, pp. 293–299, IEEE, 1993.

[26] 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*, pp. 1135–1143, 2015.

[27] 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.

[28] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Training pruned neural networks," *arXiv preprint arXiv:1803.03635*, vol. 2, 2018.

[29] A. N. Gomez, I. Zhang, S. R. Kamalakara, D. Madaan, K. Swersky, Y. Gal, and G. E. Hinton, "Learning sparse networks using targeted dropout," *arXiv preprint arXiv:1905.13678*, 2019.

[30] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

[31] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A gpu performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.

[32] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks.," *Journal of machine learning research*, vol. 10, no. 1, 2009.

[33] F. Li, "Cs231n convolutional neural networks for visual recognition lecture notes." `http://cs231n.stanford.edu/slides/2020/lecture_4.pdf`, 2020, (accessed October 15, 2020).

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[35] F. Li, "Cs231n convolutional neural networks for visual recognition." `https://cs231n.github.io/convolutional-networks/`, 2020, (accessed October 15, 2020).

[36] S. Winograd, *Arithmetic complexity of computations*, vol. 33. Siam, 1980.

[37] S. Winograd, "On multiplication of polynomials modulo a polynomial," *SIAM Journal on Computing*, vol. 9, no. 2, pp. 225–229, 1980.

[38] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[40] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[41] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning," *Coursera, video lectures*, vol. 264, no. 1, 2012.

[42] A. Graves, "Practical variational inference for neural networks," in *Advances in neural information processing systems*, pp. 2348–2356, 2011.

[43] N. Srivastava, "Improving neural networks with dropout," *University of Toronto*, vol. 182, no. 566, p. 7, 2013.

[44] 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*, vol. 15, no. 1, pp. 1929–1958, 2014.

[45] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International conference on machine learning*, pp. 1058–1066, 2013.

[46] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[47] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[48] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[49] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv preprint arXiv:1802.06367*, 2018.

[50] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[51] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *arXiv preprint arXiv:1608.03665*, 2016.

[52] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*, pp. 281–290, Springer, 2014.