Training and Pruning of Convolutional Neural Network in Winograd domain

Swastik Sahu, B. Tech.

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Data Science)

Supervisor: Dr. David Gregg

August 2019

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Swastik Sahu

August 14, 2019

Permission to Lend and/or Copy

Acknowledgments

I would like to thank my supervisor, Dr. David Gregg for his continuous support, guidance and patience. Without his support and motivation, this dissertation would not have been possible.
I am grateful to Kaveena and Andrew for sharing their knowledge with me and pointing me in the right direction by addressing my queries.
I also owe a great deal of gratitude to my parents and all those who encouraged and motivated me throughout this research.

Swastik Sahu

University of Dublin, Trinity College
August 2019

Training and Pruning of Convolutional Neural Network in Winograd domain

Swastik Sahu, Master of Science in Computer Science

University of Dublin, Trinity College, 2019

Supervisor: Dr. David Gregg

Deep convolutional neural networks can take a very long time to train on a large data-set. In critical systems like autonomous vehicles, low latency inference is desired. Most IoT and mobile devices are not powerful enough to train a successful predictive model using the data that's accessible to the device. The success of convolutional neural networks in these situations is limited by the compute power. Using Winograd's minimal filtering algorithms, we can reduce the number of multiplication operations needed for the convolution operation in spatial domain. Convolution in Winograd domain is faster when performed with small input patch and small filter kernels.

Use of Winograd techniques to perform the convolution is popular in deep learning libraries but the weights are transformed back to spatial domain after the convolution operation. In this research, we have written CPU and GPU implementations to train the weights in the Winograd domain and used different CNN architectures to train our model in spatial domain and in Winograd domain separately, on MNIST and CIFAR-10 data-set. Our GPU implementation of Winograd convolution is $\sim 2\times$ times faster than convolution in spatial domain for MNIST data-set, and $\sim 3.4\times$ times faster for CIFAR-10 data-set. Higher accuracy levels and quicker convergence was observed while training in Winograd domain for the MNIST data-set. For CIFAR-10 data-set, the effective time to converge when training in Winograd domain was $\sim 2.25\times$ times faster compared to training in spatial domain. After training separately in spatial domain and in Winograd domain, the respective weights were pruned in an iterative manner. The accuracy drop for weights trained in Winograd domain was observed to be lower than that observed for weights trained in spatial domain.

Contents

List of Tables

List of Figures

List of Abbreviations

BAIR Berkeley AI Research. 30

BVLC Berkeley Vision and Learning Center. 30

CNN Convolutional Neural Network. 1–6, 9, 10, 12, 13, 15–17, 19, 20, 22, 25, 49

CNNs Convolutional Neural Networks. 1–5, 10, 12, 13, 15, 19, 21

CPU Central Processing Unit. 3, 4, 19, 24, 34, 35, 40–47

cuFFT Cuda Fast Fourier Transformation. 10

DFT Discrete Fourier Transform. 9, 10

DRAM Dynamic Random Access Memory. 15

fbfft Facebook Fast Fourier Transformation. 10

FFT Fast Fourier Transform. 2, 9, 10, 12

GEMM General Matrix Multiply. 3, 4, 14

GPU Graphics Processing Unit. 3, 4, 12, 19, 24, 32–35, 40–46

IDFT Inverse Discrete Fourier Transform. 9

IoT Internet of Things. 1, 48

MAC Multiply Accumulate. 15

MKL	Math Kernel Library. 3, 4, 32, 34
ms	Millisecond. 41
ReLU	Rectified Linear Unit. ix, 6, 16–18
SIMD	Single Instruction Multiple Data. 13
SpMP	Sparse Matrix pre-processing. 3, 4, 31, 32, 34

Chapter 1

Introduction

Cloud computing has taken the technology world by storm, but it is not sufficient for the growing demands of the technology industry. There is already a huge amount of data that are generated at the edge devices and it is not feasible to process all the data at a central location. It is time to delegate AI capabilities at the edge and decrease dependency on the cloud. As people need to interact with their digitally-assisted technologies (e.g. wearable, virtual assistants, driverless cars, health-care, and other smart IoT devices) in real-time, waiting on a data centre far away will not work. Not only does the latency matter, but often these edge devices are not within the range of the cloud, needing them to operate autonomously. Even when these devices are connected to the cloud, moving a large amount of data to a centralized data centre is not scalable as there is a cost involved with communication and it adversely impacts the performance and energy consumption [1]. Since the latency and security risk of relying on the cloud is unacceptable, we need a large chunk of computation closer to the edge to allow secure, independent, and real-time decision making. This poses an enormous challenge in terms of implementing emerging AI workloads on resource constrained low power embedded systems. When it comes to image and video the performance of many modern embedded applications is enhanced by application of neural networks, and more specifically by convolutional neural networks (CNN).

Convolutional neural networks(CNNs) have become one of the most successful tools for applications in visual recognition. In fact, the recent and successful neural networks, like - GoogleNet [2] and ResNet [3], spend more than 90% of their time in convolutional layers. However, the training and inference tasks in CNNs are computationally expensive and the computational workload continues to grow over time as the network size keeps increasing. LeCun et al., in 1998, [4] proposed a CNN model with less than 2.3×10^7 multiplications for handwritten digit classification. Later in 2012, Krizhevsky et al. [5] developed AlexNet, an ImageNet[1]-winning CNN with more than 1.1×10^9 multiplications. In 2014, ImageNet winning and runner up CNNs increased the number of multiplications to 1.4×10^9 [2] and 1.6×10^{10} [6] respectively.

Despite the high levels of accuracy achieved, using CNNs in a low latency real-time application remains a challenge. The training and inference time in CNNs is dominated by the number of the multiplication operation in the convolution layers. Two ways to reduce this computational burden in CNNs are pruning techniques and transforming input into a different domain. Pruning introduces sparsity by removing redundant weights. Whereas, transformation techniques like Winograd convolution [7] [8] and FFT convolution [9] [10] transform the computations to a different domain where fewer multiplications are required. For an input of size N and a convolution kernel of size N, O(N^2) operations are required to perform direct convolution. Using FFT algorithms, this can be done in $O(Nlog_2N)$ operations. However, this is efficient only when the size of N is relatively large. Input size is generally large enough but the kernel is typically 3x3 or 5x5 in size. Due to the small kernel size, the constant factors can overshadow any gain in execution time. Winograd convolution performs better than FFT in practice. For an input of length l and kernel of length k, l + k -1 multiply operations are sufficient for Winograd convolution process.

With time CNNs have become deeper and even more complex. Although the accuracy of such networks have improved, some challenges that have emerged are higher latency and higher computational requirements. In this work, we explore training CNNs in Winograd domain and study the effect of simple pruning techniques.

[1]`https://en.wikipedia.org/wiki/ImageNet`

1.1 Research Question

This aim of this research is to address the following research question:

> **Can training and pruning of weights in Winograd domain improve the performance of CNNs?**

1.2 Research Objective

In order to address the research question, the aims and goals were broken down as follows:

1. Understand the fundamental concepts around deep learning, convolutional neural networks, pruning techniques, and Winograd filtering.
2. Study previous research work with respect to using Winograd transformation in CNNs.
3. Learn the fundamentals of working with a deep learning framework (TensorFlow and Caffe).
4. Learn to work with libraries (GEMM, MKL, SpMP, etc.) that optimize commonly used operations in a CNN.
5. Learn the basics of CUDA coding to implement the GPU version of Winograd convolution.
6. Implement the CPU and the GPU versions of Winograd convolution and integrate into the deep learning framework.
7. Conduct experiments with respect to training the network in the spatial domain and in the new Winograd domain, using different data-sets.
8. Conduct experiments with pruning the trained weights.

1.3 Research Challenges

1. I had to learn about a robust, production quality, deep learning framework like TensorFlow and Caffe in a short span of time. It was essential to learn to work with them because I had to integrate my work into a deep learning framework.

2. There are a lot of optimization libraries (GEMM, MKL, SpMP, etc.) used in production level deep learning frameworks. It was essential to learn the importance of these libraries and to be able to use them in my research in order to produce results that are on par with other research in this domain.

3. Having no prior CUDA coding experience, it was a challenge to implement the GPU version of the Winograd convolution layer. My goal here was to learn enough to be able to port my C++ logic in CUDA (for experiments using GPU).

4. Implementing the backward propagation logic for Winograd convolution was a challenge. There has been published research around the use of Winograd filtering algorithms for the convolution operation in CNN but I could find only one published research paper which explains the backward propagation logic to train a network in Winograd domain.

5. Time constraint was another challenge. Training a CNN on large data-sets can take up to 10-15 days of training time. It was not possible to conduct experiments on large data-sets and the scope of the research had to be re-evaluated periodically.

1.4 Dissertation Overview

Using Winograd filtering algorithms to improve the performance of CNNs is an exciting prospect. However, training the network in Winograd domain is not straight forward and very few published efforts of successfully training a network in Winograd domain to improve the performance of a CNN is available. In this research, we have implemented the CPU and the GPU version for training a CNN in Winograd domain. We have then performed different experiments on both: spatial domain & Winograd domain, using the MNIST and the CIFAR-10 data-sets. We have also performed experiments to study the effect of pruning trained weights on the performance of the network.

1.5 Dissertation Structure

The dissertation is organized as follows:

- **Chapter 2:** Discusses the recent work around optimizing the performance of CNNs, with a primary focus on work around Winograd convolution.
- **Chapter 3:** Provides the technical details of steps involved in a convolution operation, and that of our implementation of the Winograd convolution. The details of the libraries used, and our working environment is also discussed.
- **Chapter 4:** Explains the different experiments that we have performed. Details of the CNN architectures, hyper-parameters used are provided. The results of our experiments, limitations and future scope of this research are also discussed.
- **Chapter 5:** Provides a short conclusion to this research.

Chapter 2

Background and Related Work

2.1 AlexNet

AlexNet [11] is the name of a CNN, designed by Alex Krizhevsky, a PhD student at the University of Toronto in 2012. AlexNet was the winner of the ImageNet Large Scale Visual Recognition Challenge(ILSVRC)[1] 2012 with a top-5 error of 15.3%, 10.8% better than the runner up submission. It was a variation of designs proposed by Yann LeCun et al. [12] [13]. LeCun et al.'s designs were also modifications to a variant of a simpler design called *neocognitron* [14] by introducing back-propagation algorithm to it. Training AlexNet with a huge data-set like ImageNet was made feasible by the use of GPUs. There has been other research around using GPUs to leverage the performance of CNN prior to AlexNet [15] [16] but AlexNet is one of the most influential research in the computer vision and deep learning domain, especially due to its formidable performance in ImageNet Large Scale Visual Recognition Challenge 2012.

AlexNet has a very similar architecture as LeNet[13] by Yann LeCun et al. but was deeper, with more filters per layer, and with stacked convolutional layers. It has eight layers: The first five layers perform the convolution operation and some of those are followed by max-pooling layers. The last three layers are fully connected layers. ReLU layers, which performs better than tanh and sigmoid functions are used as the activation function.The original architecture of AlexNet is illustrated in figure 2.1.

[1]`http://image-net.org/about-overview`

More research around the use of CNNs in image classification challenges gave birth to other, deeper and complex, networks which improve the classification accuracy even further. We have used LeNet architecture for training our network with MNIST[2] data-set and VGG architecture for training our network with CIFAR-10[3] data-set.

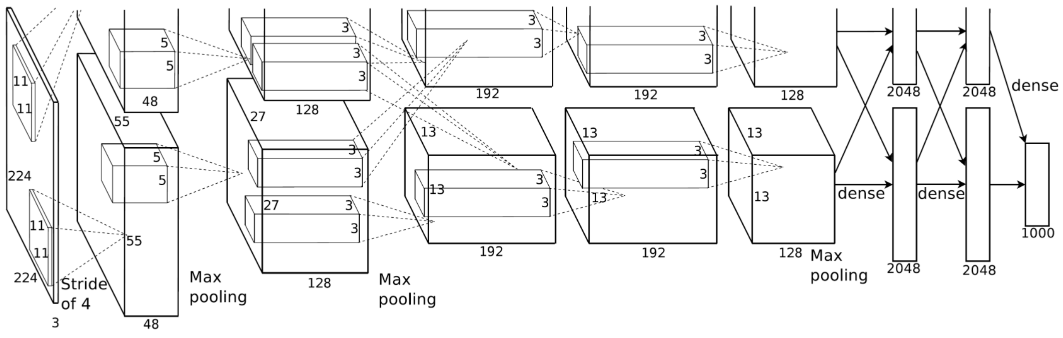


Figure 2.1: AlexNet's architecture
source: Krizhevsky et al. [11]

2.2 LeNet and VGGNet

LeNet [13], is a pioneering 7-level convolutional network proposed by LeCun et al. in 1998. It classifies digits and was used by several banks to recognize hand-written numbers on checks (cheques) digitized in 32x32 pixel grey-scale input images. The ability to process higher resolution images requires larger and more convolutional layers, so this technique was constrained by the availability of computing resources.

The runner-up at the ILSVRC 2014 competition is dubbed VGGNet [6] by the community, was developed by Simonyan and Zisserman. The original version of VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture. Similar to AlexNet, it has only 3x3 convolutions, but lots of filters. It is a popular choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many

[2]https://en.wikipedia.org/wiki/MNIST_database
[3]https://en.wikipedia.org/wiki/CIFAR-10

other applications and challenges as a baseline feature extractor. In our research, we have used a variant of VGGNet to train our network on CIFAR-10 data-set.

The convolution operation is one of the key operation in any CNN. The inputs and weights can be arranged in a manner that the convolution operation reduces to a simple matrix multiplication operation. The bulk of the workload of a convolution operation comes due to the multiplication operation. The next sections discusses some of the work that has been done to optimize the matrix multiplication and it's application to CNNs.

2.3 Strassen algorithm for fast matrix multiplication

In 1969, Volker Strassen proved that the O(N^3) time General Matrix Multiplication algorithm was not optimal and proposed a better than O(N^3) algorithm [17]. Strassen's algorithm, with running time complexity of $\mathcal{O}(n^{2.807355})$ for large input, was only slightly better but it became the basis for more research and eventually led to faster algorithms. The basis of Strassen's algorithm is explained below:
Let X, Y be two square matrices over a ring R and the product of X and Y be Z.
$\mathbf{Z} = \mathbf{XY} \qquad \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in R^{2^n \times 2^n}$

If the matrices X and Y are not of type $2^n \times 2^n$, then fill the missing rows and columns with zeros.
Then partition X, Y and Z into equally sized block matrices

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_{1,1} & \mathbf{X}_{1,2} \\ \mathbf{X}_{2,1} & \mathbf{X}_{2,2} \end{bmatrix}, \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_{1,1} & \mathbf{Y}_{1,2} \\ \mathbf{Y}_{2,1} & \mathbf{Y}_{2,2} \end{bmatrix}, \mathbf{Z} = \begin{bmatrix} \mathbf{Z}_{1,1} & \mathbf{Z}_{1,2} \\ \mathbf{Z}_{2,1} & \mathbf{Z}_{2,2} \end{bmatrix}$$

with
$\mathbf{X}_{i,j}, \mathbf{Y}_{i,j}, \mathbf{Z}_{i,j} \in R^{2^{n-1} \times 2^{n-1}} \mathbf{X}_{i,j}, \mathbf{Y}_{i,j}, \mathbf{Z}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$
The naive matrix multiplication would be:
$\mathbf{Z}_{1,1} = \mathbf{X}_{1,1}\mathbf{Y}_{1,1} + \mathbf{X}_{1,2}\mathbf{Y}_{2,1}$
$\mathbf{Z}_{1,2} = \mathbf{X}_{1,1}\mathbf{Y}_{1,2} + \mathbf{X}_{1,2}\mathbf{Y}_{2,2}$
$\mathbf{Z}_{2,1} = \mathbf{X}_{2,1}\mathbf{Y}_{1,1} + \mathbf{X}_{2,2}\mathbf{Y}_{2,1}$
$\mathbf{Z}_{2,2} = \mathbf{X}_{2,1}\mathbf{Y}_{1,2} + \mathbf{X}_{2,2}\mathbf{Y}_{2,2}$

8 multiplications to calculate the $Z_{i,j}$ values, which is the same as in standard matrix multiplication.
The Strassen algorithm suggests defining new matrices as below:
$\mathbf{M}_1 := (\mathbf{X}_{1,1} + \mathbf{X}_{2,2})(\mathbf{Y}_{1,1} + \mathbf{Y}_{2,2})$
$\mathbf{M}_2 := (\mathbf{X}_{2,1} + \mathbf{X}_{2,2})\mathbf{Y}_{1,1}$
$\mathbf{M}_3 := \mathbf{X}_{1,1}(\mathbf{Y}_{1,2} - \mathbf{Y}_{2,2})$
$\mathbf{M}_4 := \mathbf{X}_{2,2}(\mathbf{Y}_{2,1} - \mathbf{Y}_{1,1})$
$\mathbf{M}_5 := (\mathbf{X}_{1,1} + \mathbf{X}_{1,2})\mathbf{Y}_{2,2}$
$\mathbf{M}_6 := (\mathbf{X}_{2,1} - \mathbf{X}_{1,1})(\mathbf{Y}_{1,1} + \mathbf{Y}_{1,2})$
$\mathbf{M}_7 := (\mathbf{X}_{1,2} - \mathbf{X}_{2,2})(\mathbf{Y}_{2,1} + \mathbf{Y}_{2,2})$
With M_i defined as above, only 7 multiplications (one for each M_i) are required. $C_{i,j}$ can now be expressed in terms of M_i as follows:
$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$
$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$
$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$
$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$
This division process is repeated n times recursively until the sub-matrices degenerate into numbers (elements of the ring R). At this point the final product is padded with zeroes, just like X and Y, and is stripped of the corresponding rows and columns.

Using this basis, Cong and Xiao [18] proposed a view of the CNN architecture that can leverage from Strassen's algorithm. They reported to have reduced the computation by up to 47%.

More research in optimizing the performance of CNNs led to exploration of domain transformation techniques such as FFT, details of which are discussed in the next section 2.4.

2.4 FFT based convolution

A fast Fourier transform (FFT) is a technique that calculates the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis involves converting a signal from a base domain(space/time) to a representation in frequency domain and vice-versa. The DFT is computed by breaking a series of values into components of distinct frequencies [19]. This operation can be applied to many fields but computing the DFT requires too many operations to be feasible in practice. An FFT performs the conversion by factorizing the DFT input into a product of sparse factors in faster time [20]. Thus the time complexity of calculating the DFT reduces from O(N^2), which arises when one uses the definition of DFT, to O($Nlog_2N$). Here N represents the size of the input.

In 2014, Mathieu et al. [10] presented a simple FFT based convolution algorithm to accelerate the training and inference of a CNN. Their FFT based CNN outperformed the state-of-the-art CNN implementations of that time. This was possible by computing convolutions as Hadamard products[4] in the Fourier domain and reusing the same transformed feature map.

In 2015, Vasilache et al. [9] proposed two FFT based convolution techniques to optimize the performance of CNNs. One of their implementations was based on NVIDIA's cuFFT library, and the other was based Facebook's open-source library, fbfft. Both for them were faster than NVIDIA's CuDNN implementation of networks with many convolution layers. fbfft performed better than cuFFT, however, the speedups were prominent for deeper networks and large kernel sizes. The speedups obtained by their cuFFT implementation for different kernel sizes are shown in figure 2.2.

The speedups achieved using FFT based convolution are promising but are observed for large input and kernel sizes. The typical kernel sizes used in CNNs are small (3×3 or 5×5) and FFT based convolution techniques have produced mixed results with small kernel sizes.

2.5 Winograd based convolution

Like FFT, Winograd based algorithms are another class of transformation techniques that use properties of linear algebra to reduce the number of multiplication operation. Coppersmith-Winograd algorithm was one of the earliest version of this technique and has a running time of $\mathcal{O}(n^{2.375477})$ [21], for multiplying two square matrices of size $n \times n$. Unlike FFT based techniques, Winograd convolution works well with small input and kernel sizes, which is generally the case in a CNN. Using Winograd's minimal filtering algorithm-based optimization techniques is a common practice in signal processing and

[4]`https://en.wikipedia.org/wiki/Hadamard_product_(matrices)`

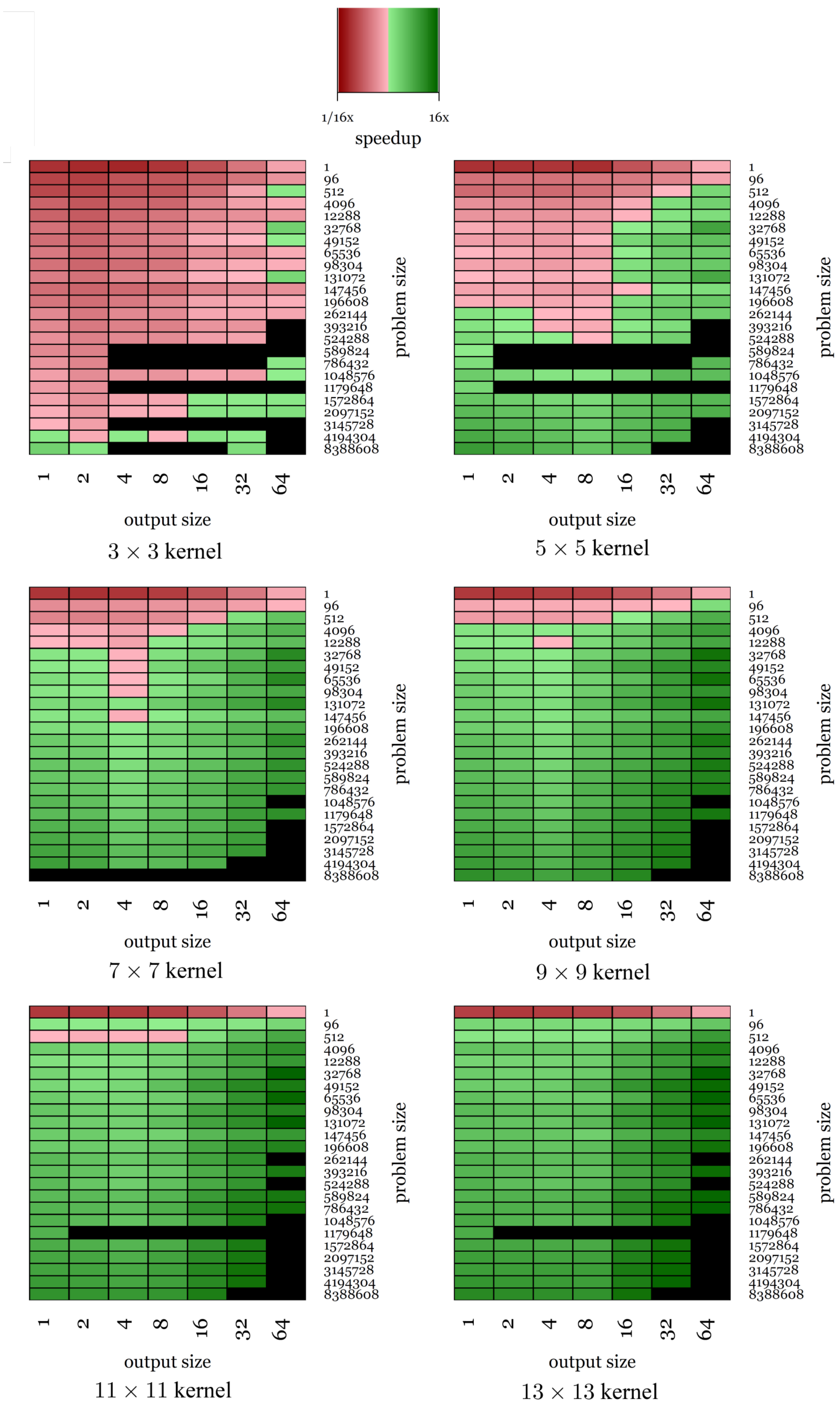


Figure 2.2: Speed ups for cuFFT convolution by Vasilache et al. for different kernel sizes

source: Vasilache et al. [9]

data transmission but wasn't used in CNN until 2015.

Lavin and Gray [8] were the first to propose an analogy between the convolution operation in CNN and application of Winograd's minimal filtering techniques. They implemented the GPU version of Winograd based convolution layer on the VGG Network [6], a sixteen layer deep network with nine convolution layers. They performed bench-marking tests for different batch sizes with both single precision (fp32) and half precision (fp16), using a 3×3 filter on every convolution layer. fp32 arithmetic instructions were used for all the tests. With fp32 data, they found $F(2 \times 2, 3 \times 3)$ based Winograd convolution to be more accurate than direct convolution. With fp16 data, all the algorithms were observed to produce similar accuracy. Speedups were observed for both fp16 and fp32 data using Winograd based convolution.

After the work by Lavin and Gray [8], the use of Winograd based convolutions gained popularity and a standard implementation of it was provided on various hardware platforms. Compared to spatial or FFT based convolution, fewer floating-point operations are involved in Winograd based convolution and they are very popular in CNNs. However, training CNNs in Winograd domain is not straight forward. Winograd based convolutions are used only to perform the convolution operation and the weights are converted back to spatial domain after the convolution operation.

Zlateski et al. [22] studied the behaviour of these convolution techniques(regular FFT, Gaussian FFT, and Winograd) on modern CPUs. Their evaluation criteria was based on experiments using VGG [6] and AlexNet [11] and took memory bandwidth and cache sizes, along with floating point operations, into consideration. They found FFT based convolution techniques to be faster than Winograd based method on multi core CPUs with large cache sizes.

We have also used Winograd based techniques for convolution in our CNNs for this research. The details of the transformation and convolution operation in Winograd domain, and those for the backward propagation stage are discussed in Chapter 3.

2.6 Winograd Convolution via Integer Arithmetic

Quantized CNNs have been shown to work for inference with integer weights and activations [23]. The 32-bit floating point model sizes can be reduced by a factor of 4 by quantizing to 8-bit integers. The quantized networks on CPUs are also $2\times$-$3\times$

times faster than the floating point models. Various successful kernels like ARM CMSIS [24], GEMMLOWP (GLP), Nvidia Tensor RT use reduced precision for fast inference. There also exist a few custom hardware that use reduced precision for fast inference [25, 1]

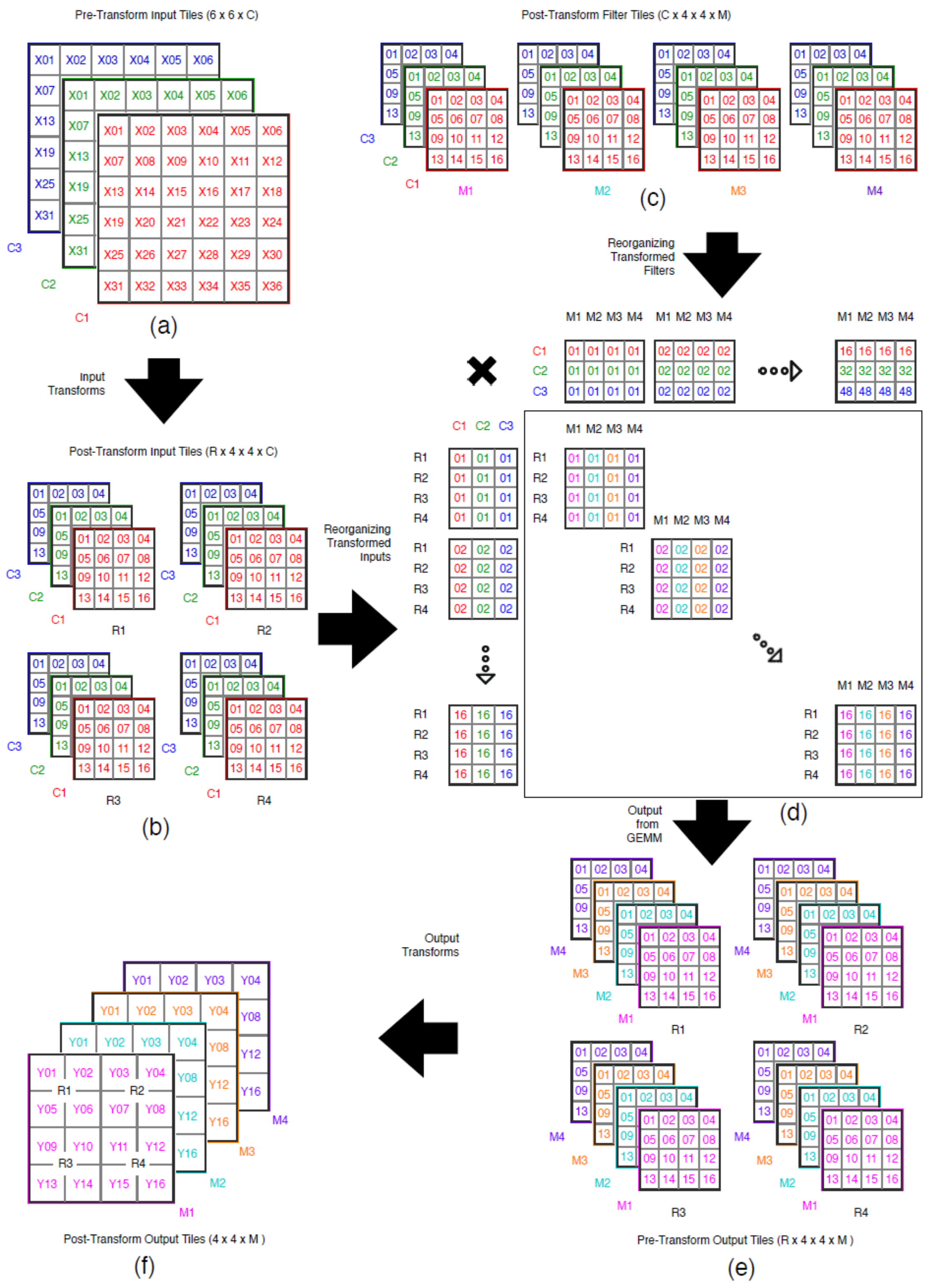
Meng and Brothers [26] proposed a new Winograd based convolution by extending the construction to the field of complex. They have reported their approach to attain an arithmetic complexity reduction of $4\times$ over the spatial convolution and $2\times$ performance gain over other algorithms. They also propose an integer-based filter scaling scheme that reduces the kernel bit width by 2.5% without much accuracy loss.They displayed that a mixture of Winograd based convolution and lossy scaling scheme can attain inference speedups without much accuracy loss.

2.7 Winograd Convolution Kernel Implementation on embedded CPUs(ARM architecture)

One of the primary goals of improving the performance of CNNs using Winograd and other transformation techniques is to enable deployment in low power embedded devices. Maji et al. [27] propose a way to leverage computational gains on an Armv8-A architecture by rearranging blocks of data and allow optimal use of the available SIMD registers. The proposed data flow is described in figure 2.3. They reported having achieved 30% - 60% speedups in deep CNNs like: SqueezeNet [28], Inception-v3 [29], GoogleNet [2] and VGG16 [6]. In theory, speedups up to $4\times$ are possible using this approach. However, the practical speedups are lower than this, largely due to the cost associated with transforming the inputs. With an increase in the number of output channels, the speedups attained will be maximized.

2.8 Pruning techniques

The complexity of a CNN architecture depends on the problem it has to solve. For a simple task, a shallow network with few parameters may be sufficient, whereas for a complex task a deep network with several connections and parameters may be required. The computational complexity increases as the size of the network increases. A com-



Pre-Transform Input Tiles (6 x 6 x C)
(a)
Input Transforms
Post-Transform Input Tiles (R x 4 x 4 x C)
(b)
Post-Transform Filter Tiles (C x 4 x 4 x M)
(c)
Reorganizing Transformed Filters
Reorganizing Transformed Inputs
(d)
Output from GEMM
Output Transforms
Post-Transform Output Tiles (4 x 4 x M)
(f)
Pre-Transform Output Tiles (R x 4 x 4 x M)
(e)

Figure 2.3: Data-Flow in Region-wise Multi-channel on ARM CPUs
(a) Pre-transform Input Channels, (b) Transformed Input, (c) Transformed Filters, (d) GEMM Kernels, (e) Output of GEMM in the Residue Domain, (f) Final Output Channels after applying Inverse Transforms
source: Maji et al. [27]

plex network is capable of learning a lot of details, sometimes more than what's necessary for the given task. The optimal configuration is unknown and can require a bit of guess work. Learning more parameters than necessary may lead to over-fitting, whereas learning less than necessary may lead to under-fitting. One solution to this problem is to prune a superior network by discarding redundant connections and useless weights [30, 31]. This works well in most cases and provides better results. A different issue while dealing with complex CNNs is with porting them on low power embedded devices. Here again, we can solve part of the problem by deploying a pre-trained *pruned* network on the device. This will reduce the DRAM accesses and in turn save energy. As CNNs can perform several MAC operations, sparsity can help in reducing those computations.

Several network pruning methods have been shared by different researchers. Han, Pool et al. [32] and Han, Mao et al. [33] were able to discard a large portion of the weights with minimum accuracy drop by training the network with L1/L2 norm augmented loss functions and pruning it gradually. Connection values less than a threshold are removed and the threshold is increased gradually. Han, Pool et al. [32] extended their work by quantizing the final pruned network[33]. Han, Pool et al. [32] and Han, Mao et al. [33] had to use sparse representation to leverage from the induced sparsity. Han et al. [34, 33] proposed learning the sparsity pattern in network weights by discarding weights whose absolute value is below a certain threshold. This technique can induce 50%-70% sparsity in the network and reduce the number of multiplications. Castellano et al. pruned the units in the hidden layers for a feed-forward deep CNN [35].

Collins and Kohli reduce the computational workload with sparse connectivity in convolution and fully connected layers [36]. Stepniewski and Keane [31] used genetic algorithms and simulated annealing to prunes multi-layered perceptron. These works [32, 33, 35, 36, 31] leverage from the unstructured sparsity in CNN.

Polyak and Wolf [37] propose a way to add channel-wise sparsity to a network by removing entire convolution layer channels. On the other hand, Anwar et al. in proposed a way to explore sparsity at different levels using search followed fixed point optimization [38].

In Dropout [39] and Dropconnect [40] the neuron output and weights are pruned during the training phase. Both of these methods train discrete subgroups of network parameters and produce better generalizations. Louizos et al. [41] use a variant of concrete dropout [42] on the weights of a network and regularize the drop rates in order to make the network sparse. Likewise, Molchanov et al. [43] reported that applying variational dropout [44] to the weights of a network implicitly sparsifies the parameters.

Liu et al. [45] were among the first to propose pruning and retraining the weights in Winograd domain for Winograd based convolution in CNN. Li et al. [46] also reported promising results on large data-sets. They reported that approximation techniques, used on smaller networks by Liu and Turakhia [47], weren't sufficient to handle non-invertible mapping between convolution and Winograd parameters on larger networks like AlexNet [5]. They reported having achieved 90% sparsity in the Winograd parameters of AlexNet [5] with accuracy drop as low as 0.1%.

There has been much research which suggests the use of pruning and retraining [46, 45] to regain the accuracy drop. In this research, we have trained our models in the spatial domain and the Winograd domain separately, then pruned the respective trained weights in an iterative manner to study the drop in accuracy.

Pruning methods can be applied with dynamic activation sparsity to further improve the multiplication workload. Some of the previous work using dynamic activation sparsity to improve CNN's performance is discussed in the next section 2.9.

2.9 Dynamic Activation Sparsity

The ReLU(rectified linear unit), defined as below sets activations whose values are negative to zero.
$f(x) = x^+ = \max(0, x)$
This induces dynamic sparsity in the activations. Han et al. [34] reported that exploring sparsity of both weights and activations can lower the count of multiplication operation by $4 - 11\times$. Huan et al. [48] further experimented and reported that changing the ReLU configurations after training the network can induce greater sparsity in activation without any significant accuracy loss. Exploring unconventional CNN architectures has also led to improvements for deep learning accelerators to take advantage of the sparsity in activations. Han et al. [25] reported having optimally skip zeros in input activations by deploying a *Leading Non-zero Detection unit*(LNZD) in their

accelerator. Albericio et al. [49] also suggested a similar technique for a convolution layer accelerator.

Transformation techniques and pruning methods are two independent ways of optimizing a CNN. Combining these two approaches can help in optimizing the network even more. However, combining these two approaches is not straight forward as conventional data transformation neutralizes any gains obtained from the sparse methods. Figure 2.4 illustrates three different strategies for using Winograd based data transformation techniques in CNN. Liu et al. [45] proposed a Winograd-ReLU base CNN by moving the ReLU layer after the Winograd transformation. Both ReLU and pruning were performed after Winograd transformation in this architecture(figure2.4c). Using this approach they were able to reduce the count of multiplication operation by $10.4\times, 6.8\times$ and $10.8\times$ with less than 0.1% accuracy drop while training with CIFAR-10, CIFAR-100 and ImageNet data-sets respectively. They also reported that this architecture allows for more aggressive pruning.

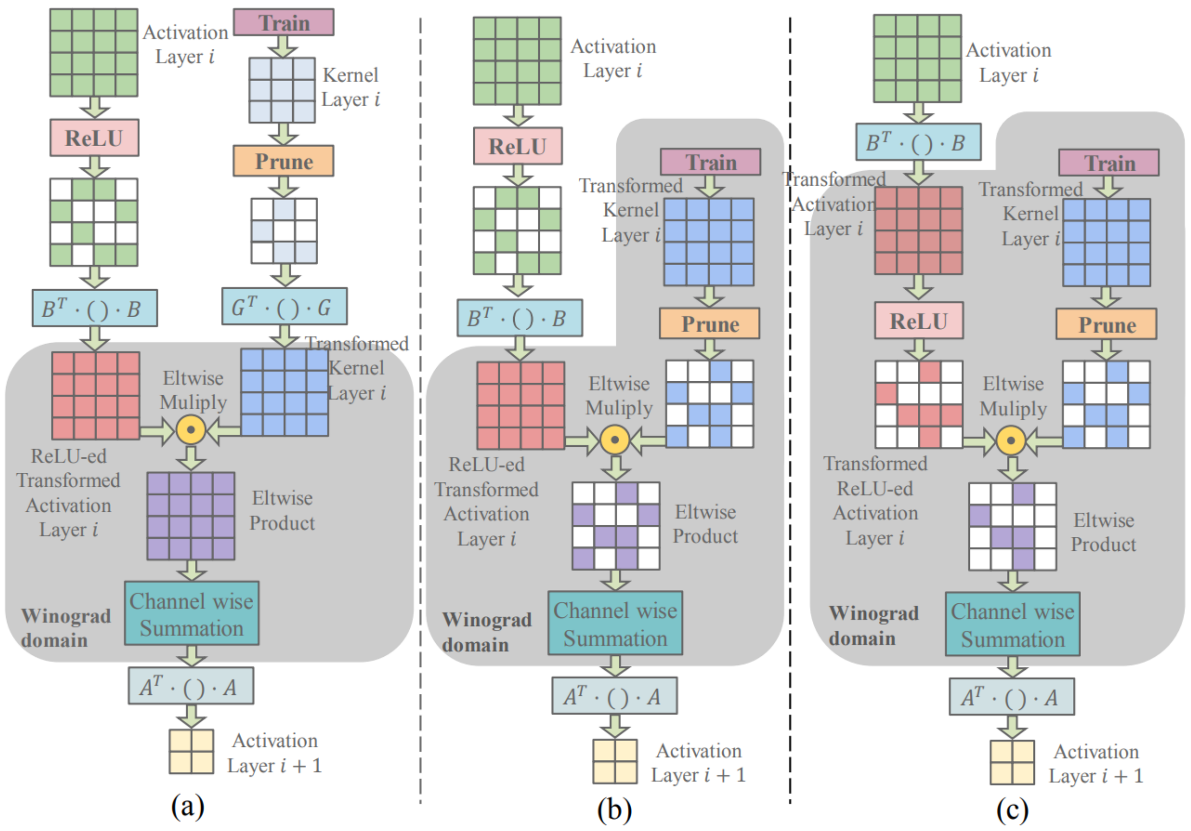


Figure 2.4: Winograd-ReLU
(a) Standard Winograd based convolution fills the 0s in both the weights and activations. (b) Pruning the transformed filter restores sparsity to the weights. (c) Winograd-ReLU based architecture restores sparsity to both the weights and activations. source: Liu et al. [45]

Chapter 3

Convolution in CNN

One of the primary task in this research was to integrate a Winograd convolution layer in the CNN architecture where the weights are trained in the Winograd domain. Section 3.1 of this chapter provides a brief overview of convolution operation and the back-propagation logic in spatial domain. In section 3.2, the concepts involved in Winograd minimal filtering techniques are discussed and an analogy of convolution operation in CNNs & 2D Winograd minimal filtering technique is provided. The steps involved in gradient computation in the Winograd domain are also discussed in 3.2. Finally, the CPU/GPU implementation details of our Winograd convolution layer is discussed in section 3.3.

3.1 Spatial Convolution

In CNNs, the input is an image with height H and width W. Depending on whether the image is grey-scale or coloured, it can have multiple channels C. So one image has H×W×C data points. A kernel is used as a filter to learn different features about the input image. Typically, small kernel sizes of 3×3×C or 5×5×C are used. Large kernel will not be able to capture minute details and hence result in low accuracy.

Let's assume an input of size 32×32×3 and a 5×5×3 kernel filter. If we slide it over the complete image and along the way take the dot product between the filter and chunks of the input image, we get a 28×28×1[1] output(figure 3.1). This is

[1]In practice, most libraries pad the input matrix with zero to produce a 32x32x1 output.

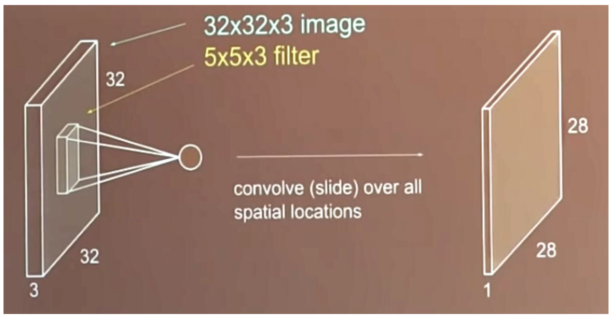


Figure 3.1: Convolution
image source: Internet

what happens in the forward pass of the convolution layer in the CNN. The convolution layer comprises of a set of independent filters, which are independently convoluted with the image. The feature maps *learn* certain characteristics(edge, sharpness, etc.) of the input image.

In the forward pass the feature maps are generated by convolution of the image and the kernel. The loss function is calculated from this value. Then in the backward pass, the weights in each of the layers are adjusted to minimize the loss in reverse order - from the last layer to the first layer. A simple illustration of convolution(during forward pass) and back-propagation(during backward pass) is presented in figure 3.2, where $\mathbf{X} = Input; \mathbf{F} = Kernel; \mathbf{O} = Output;$

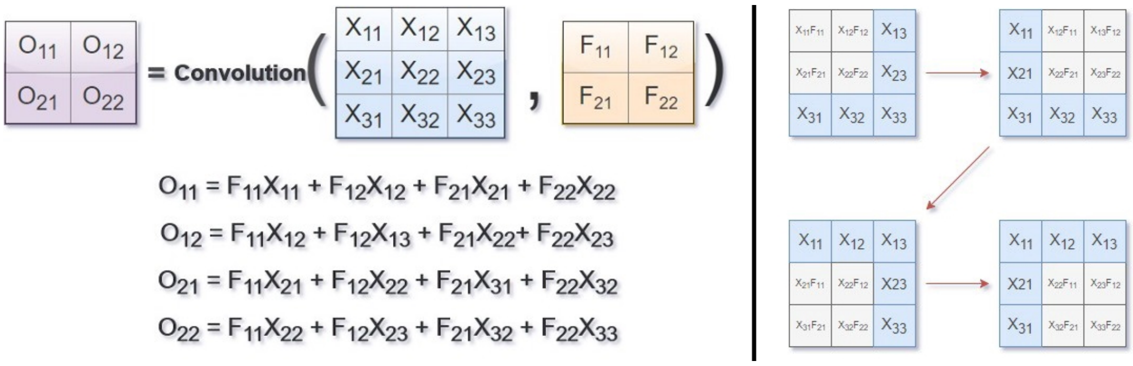

O_{11} O_{12} O_{21} O_{22} = Convolution (X_{11} X_{12} X_{13} X_{21} X_{22} X_{23} X_{31} X_{32} X_{33} , F_{11} F_{12} F_{21} F_{22})
$O_{11} = F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22}$
$O_{12} = F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23}$
$O_{21} = F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32}$
$O_{22} = F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33}$

Figure 3.2: Forward Pass
image source: Internet

In the backward pass, the gradients of the kernel 'F' with respect to the

error $'E'$ is computed following equations shown in figure 3.3.

$$
\begin{aligned}
\frac{\partial E}{\partial F_{11}} &= \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{11}} \\
\frac{\partial E}{\partial F_{12}} &= \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{12}} \\
\frac{\partial E}{\partial F_{21}} &= \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{21}} \\
\frac{\partial E}{\partial F_{22}} &= \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{22}}
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\frac{\partial E}{\partial F_{11}} &= \frac{\partial E}{\partial O_{11}}X_{11} + \frac{\partial E}{\partial O_{12}}X_{12} + \frac{\partial E}{\partial O_{21}}X_{21} + \frac{\partial E}{\partial O_{22}}X_{22} \\
\frac{\partial E}{\partial F_{12}} &= \frac{\partial E}{\partial O_{11}}X_{12} + \frac{\partial E}{\partial O_{12}}X_{13} + \frac{\partial E}{\partial O_{21}}X_{22} + \frac{\partial E}{\partial O_{22}}X_{23} \\
\frac{\partial E}{\partial F_{21}} &= \frac{\partial E}{\partial O_{11}}X_{21} + \frac{\partial E}{\partial O_{12}}X_{22} + \frac{\partial E}{\partial O_{21}}X_{31} + \frac{\partial E}{\partial O_{22}}X_{32} \\
\frac{\partial E}{\partial F_{22}} &= \frac{\partial E}{\partial O_{11}}X_{22} + \frac{\partial E}{\partial O_{12}}X_{23} + \frac{\partial E}{\partial O_{21}}X_{32} + \frac{\partial E}{\partial O_{22}}X_{33}
\end{aligned}
$$

Figure 3.3: Backward Pass

The above equation can be written in form of a convolution operation as shown in figure 3.4.

$$
\begin{bmatrix} \partial E/\partial F_{11} & \partial E/\partial F_{12} \\ \partial E/\partial F_{21} & \partial E/\partial F_{22} \end{bmatrix}
= \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix} \right)
$$

Figure 3.4: Backward Pass
image source: Internet

Likewise, the gradient computation for the input X can be summarize as shown in figure 3.5:

$$
\begin{bmatrix} \partial E/\partial X_{11} & \partial E/\partial X_{12} & \partial E/\partial X_{13} \\ \partial E/\partial X_{21} & \partial E/\partial X_{22} & \partial E/\partial X_{23} \\ \partial E/\partial X_{31} & \partial E/\partial X_{32} & \partial E/\partial X_{33} \end{bmatrix}
= \text{Full\_Convolution}\left(\begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix}, \begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix} \right)
$$

Figure 3.5: Backward Pass
image source: Internet

3.2 Winograd Convolution

Lavin and grey were the first to apply Winograd minimal filtering techniques for convolution operation in CNNs. The details suggested by them in their paper [8] are discussed in this section.

A convolution layer in CNN processes:
(i) a bank of K filters with C channels and shape of $R \times S$, with
(ii) a batch of N images with C channels, height H and width W
The filter elements can be denoted as $G_{k,c,u,v}$, and
the input images can be denoted as $D_{i,c,x,y}$.
The output of a single convolution layer can then be denoted by:

$$Y_{i,k,x,y} = \sum_{c=1}^{C}\sum_{v=1}^{R}\sum_{u=1}^{S} D_{i,c,x+u,y+v}G_{k,c,u,v} \tag{3.1}$$

which can be written as:

$$Y_{i,k} = \sum_{c=1}^{C} D_{i,c} * G_{k,c} \tag{3.2}$$

where $*$ denotes 2D correlation.
Using Winograd's minimal filtering algorithm for computing m outputs with an r-tap FIR filter, F(m, r), requires:

$$\mu(\ F(m,r)\ )\ =\ m\ +\ r\ -\ 1 \tag{3.3}$$

multiplications [7]. The 1D algorithm can be nested to form 2D algorithms. So the number of multiplication required for computing o$m \times n$ outputs with an $r \times s$ filter, F(m×n, r×s) is given by [7]:

$$\mu(F(m\times n, r\times s)) \ = \ \mu(F(m,r))\mu(F(n,s)) \ = \ (m\ +\ r\ -\ 1)(n\ +\ s\ -\ 1) \tag{3.4}$$

Equations 3.3 & 3.4 show that to compute $F(m,r)$, we must access an interval of $m + r - 1$ data values, and to compute $F(m \times n, r \times s)$ we must access a tile of $(m+r-1)\times(n+s-1)$ data values. So one multiplication per input is required when using minimum filtering algorithm.

$\mathbf{F(2\times 2, 3\times 3)}$

The standard algorithm for F(2, 3) uses 2×3 = 6 multiplications. Winograd [7] documented the following minimal algorithm:

$$\mathbf{F(2,3)} = \begin{bmatrix} \mathbf{d}_0 & \mathbf{d}_1 & \mathbf{d}_2 \\ \mathbf{d}_1 & \mathbf{d}_2 & \mathbf{d}_3 \end{bmatrix} \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \mathbf{g}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{m_1 + m_2 + m_3} \\ \mathbf{m_2 - m_3 - m_4} \end{bmatrix} \tag{3.5}$$

where
$m_1 = (d_0 d_2) g_0$; $\quad m_2 = (d_1 + d_2)\frac{(g_0+g_1+g_2)}{2}$; $\quad m_3 = (d_2 d_1)\frac{g_0 g_1 + g_2}{2}$; $\quad m_4 = (d_1 d_3) g_2$

Using equation 3.3, number of multiplications in F(2, 3) $= \mu(F(2,3)) = 2 + 3 - 1 = 4$. It also uses 4 additions involving the data, 3 additions and 2 multiplications by a constant involving the filter (the sum $g_0 + g_2$ can be computed just once), and 4 additions to reduce the products to the final result.
Fast filtering algorithms can be written in matrix form as:

$$Y = A^T[(Gg) \odot (B^T d)] \tag{3.6}$$

where $\odot$ indicates element-wise multiplication. For F(2, 3), **B, G, A, g** & **d** are given by [7]: $\mathbf{B^T} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & -\mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & -\mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & -\mathbf{1} \end{bmatrix}$ $\quad \mathbf{G} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \frac{\mathbf{1}}{\mathbf{2}} & \frac{\mathbf{1}}{\mathbf{2}} & \frac{\mathbf{1}}{\mathbf{2}} \\ \frac{\mathbf{1}}{\mathbf{2}} & -\frac{\mathbf{1}}{\mathbf{2}} & \frac{\mathbf{1}}{\mathbf{2}} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$ $\quad \mathbf{A^T} = \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & -\mathbf{1} & -\mathbf{1} \end{bmatrix}$

$$\mathbf{g} = \begin{bmatrix} \mathbf{g}_0 & \mathbf{g}_1 & \mathbf{g}_2 \end{bmatrix}^T \quad \mathbf{d} = \begin{bmatrix} \mathbf{d}_0 & \mathbf{d}_1 & \mathbf{d}_2 & \mathbf{d}_3 \end{bmatrix}^T \tag{3.7}$$

The 2D filtering algorithm can be obtained by nesting the 1D algorithm to itself:

$$Y = A^T[[GgG^T] \odot [B^T dB]]A \tag{3.8}$$

where now g is an $r \times r$ filter and d is an $(m + r - 1) \times (m + r - 1)$ image tile. The nesting technique can be generalized for non-square filters and outputs, $F(m \times n, r \times s)$, by nesting an algorithm for F(m, r) with an algorithm for F(n, s).

$F(2\times2, 3\times3)$ uses $4\times4 = 16$ multiplications, whereas the standard algorithm uses $2 \times 2 \times 3 \times 3 = 36$ multiplications. This is an arithmetic complexity reduction of $\frac{36}{16}$

= 2.25. The data transform uses 32 additions, the filter transform uses 28 floating point instructions, and the inverse transform uses 24 additions.

Algorithms for F(m×m,r×r) can be used to compute convolution layers with r×r kernels. Each image channel is divided into tiles of size (m+r1)×(m+r1), with $r-1$ elements of overlap between neighboring tiles, yielding $P = \lceil \frac{H}{m} \rceil \lceil \frac{W}{m} \rceil$ tiles per channel, C. F(m×m,r×r) is then computed for each tile and filter combination in each channel, and the results are summed over all channels. Substituting $U = GgG^T andV = B^T dB$, we have:

$$Y = A^T[U \odot V] \tag{3.9}$$

Labeling tile coordinates as ($\tilde{x}$,$\tilde{y}$), the convolution layer formula 3.2 for a single image i, filter k, and tile coordinate ($\tilde{x}$,$\tilde{y}$) can be written as:

$$Y_{i,k,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} D_{i,c,\tilde{x},\tilde{y}} * G_{k,c} = \sum_{c=1}^{C} A^T[U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}}]A = A^T[\sum_{c=1}^{C} U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}}]A \tag{3.10}$$

Thus we can reduce over C channels in transform space, and only then apply the inverse transform A to the sum. This amortizes the cost of the inverse transform over the number of channels.
The following sum:

$$M_{k,i,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}} \tag{3.11}$$

can be simplified by collapsing the image/tile coordinates ($i, \tilde{x}$,$\tilde{y}$) down to a single dimension, b. If we label each component the element-wise multiplication separately, as (ξ, ν), we have:

$$M_{k,b}^{(\xi,\nu)} = \sum_{c=1}^{C} U_{k,c}^{(\xi,\nu)} V_{c,b}^{(\xi,\nu)} \tag{3.12}$$

This is just a basic matrix multiplication and can be written as:

$$M^{(\xi,\nu)} = \sum_{c=1}^{C} U^{(\xi,\nu)} V^{(\xi,\nu)} \tag{3.13}$$

Many efficient implementations of matrix multiplication are available on CPU, GPU, and FPGA platforms. Thus using the above equations as basis, an algorithm to compute the convolution operation in the forward pass of a CNN is presented in section 3.2.1.

Winograd documented a technique for generating the minimal filtering algorithm $F(m, r)$ for any choice of m and r. The construction uses the Chinese remainder theorem to produce a minimal algorithm for linear convolution, which is equivalent to polynomial multiplication, then transposes the linear convolution algorithm to yield a minimal filtering algorithm. More details are present in Winograd's seminal book [7].

$\mathbf{F(3 \times 3, 2 \times 2)}$

Training a network using stochastic gradient descent requires computation of the gradients with respect to the inputs and weights. For a convolution layer, the gradient with respect to the inputs is a convolution of the next layers back-propagated error, of dimension N×K×H×W, with a flipped version of the layers R×S filters. Therefore it can be computed using the same algorithm that is used for forward propagation. The gradient with respect to the weights is a convolution of the layer inputs with the back-propagated errors, producing R×S outputs per filter and channel. Therefore we need to compute the convolution F(R×S,H×W), which is impractical because H×W is too large for our fast algorithms. Instead decomposing this convolution into a direct sum of smaller convolutions [8], for example F(3×3, 2×2). Here the algorithm's 4×4 tiles are overlapped by 2 pixels in each dimension, and the 3×3 outputs are summed over all tiles to form F(3×3,H ×W). The transforms for F(3×3, 2×2) are given by:

$$\mathbf{B^T} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & -1 & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & -1 & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & -1 & \mathbf{0} & \mathbf{1} \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \frac{\mathbf{1}}{\mathbf{2}} & \frac{\mathbf{1}}{\mathbf{2}} \\ \frac{\mathbf{1}}{\mathbf{2}} & -\frac{\mathbf{1}}{\mathbf{2}} \\ \mathbf{0} & \mathbf{1} \end{bmatrix} \quad \mathbf{A^T} = \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & -1 & \mathbf{0} \\ \mathbf{0} & -1 & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix} \tag{3.14}$$

With $(3{+}21)^2 = 16$ multiplies versus direct convolutions $3{\times}3{\times}2{\times}2 = 36$ multiplies, it achieves the same $36/16 = 2.25$ arithmetic complexity reduction as the corresponding forward propagation algorithm.

F(4 × 4, 3 × 3)

A minimal algorithm for F(4, 3) has the form:

$$\mathbf{B^T} = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{A^T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \tag{3.15}$$

The data transform uses 13 floating point instructions, the filter transform uses 8, and the inverse transform uses 10.

Applying the nesting formula yields a minimal algorithm for F(4×4, 3×3) that uses 6×6 = 36 multiplies, while the standard algorithm uses 4×4×3×3 = 144. This is an arithmetic complexity reduction of 4. The 2D data transform uses 13(6 + 6) = 156 floating point instructions, the filter transform uses 8(3 + 6) = 72, and the inverse transform uses 10(6 + 4) = 100.

The number of additions and constant multiplications required by the minimal Winograd transforms increases quadratic-ally with the tile size [7]. Thus for large tiles, the complexity of the transforms will overwhelm any savings in the number of multiplications.

The magnitude of the transform matrix elements also increases with increasing tile size. This effectively reduces the numeric accuracy of the computation, so that for large tiles, the transforms cannot be computed accurately [7].

3.2.1 Forward Propagation

In the forward pass an input image of size $H \times W$ is convoluted with a kernel of size $R \times S$. In practice, the size of kernel is small but the image size can vary (from smaller than 32 × 32 to larger than 224 × 224). In theory, it is possible to compute $F(R \times S, H \times W$ using methods mention in Winograd's seminal book [7]. However the complexity to compute $F(R \times S, H \times W)$ increases quadratic-ally and it is not practical to compute $F(R \times S, H \times W)$ for large $H \times W$. Instead, the input image is broken into smaller segments(tiles) and the Winograd convolution is performed on these tiles.

Using equations 3.1 - 3.13, the convolution operation in Winograd domain can be summarized as algorithm 3.6:

Algorithm 1 Compute Convnet Layer with Winograd Minimal Filtering Algorithm $F(m \times m, r \times r)$

```
$P = N\lceil H/m \rceil \lceil W/m \rceil$ is the number of image tiles.
$\alpha = m + r - 1$ is the input tile size.
Neighboring tiles overlap by $r - 1$.
$d_{c,b} \in \mathbb{R}^{\alpha \times \alpha}$ is input tile $b$ in channel $c$.
$g_{k,c} \in \mathbb{R}^{r \times r}$ is filter $k$ in channel $c$.
$G$, $B^T$, and $A^T$ are filter, data, and inverse transforms.
$Y_{k,b} \in \mathbb{R}^{m \times m}$ is output tile $b$ in filter $k$.
for $k = 0$ to $K$ do
    for $c = 0$ to $C$ do
        $u = G g_{k,c} G^T \in \mathbb{R}^{\alpha \times \alpha}$
        Scatter $u$ to matrices U: $U_{k,c}^{(\xi,\nu)} = u_{\xi,\nu}$
for $b = 0$ to $P$ do
    for $c = 0$ to $C$ do
        $v = B^T d_{c,b} B \in \mathbb{R}^{\alpha \times \alpha}$
        Scatter $v$ to matrices V: $V_{c,b}^{(\xi,\nu)} = v_{\xi,\nu}$
for $\xi = 0$ to $\alpha$ do
    for $\nu = 0$ to $\alpha$ do
        $M^{(\xi,\nu)} = U^{(\xi,\nu)} V^{(\xi,\nu)}$
for $k = 0$ to $K$ do
    for $b = 0$ to $P$ do
        Gather m from matrices M: $m_{\xi,\nu} = M_{k,b}^{(\xi,\nu)}$
        $Y_{k,b} = A^T m A$
```

Figure 3.6: Winograd Convolution

3.2.2 Backward Propagation

Use of Winograd filtering techniques suggested by Lavin and grey [8] leverages from the reduction in multiplication operation by applying Winograd convolution on both forward and backward propagation. However, to truly train a network in the Winograd domain, the gradient must be computed with respect to the transformed output. Li et al. in their paper [46] had proposed a way to compute the gradients with respect to the loss function, in Winograd domain. The details of the back-propagation logic are discussed in this section.

Let's assume a convolution layer where the input tensors with C channels of features maps each of dimension $H_i \times W_i$ is transformed into K output channels via a simple unit-stride, unit-dilation linear convolution with kernels of size $r \times s$:

$$\begin{aligned} I \in \mathbb{R}^{C\times H_i\times W_i} &\longrightarrow O \in \mathbb{R}^{K\times H_o\times W_o} \\ via \qquad O(k,:,:) &= \sum_{c=1}^{C} W(k,c,:,:) \odot I(c,:,:) \end{aligned} \tag{3.16}$$

where $W_o = W_i - s + 1$, $H_o = H_i - r + 1$, and $\odot$ stands for 2D linear convolution. The computation of Equation 3.16 can be performed using the Winograd transform which has a lower arithmetic complexity than Equation 3.16 suggests. A convolution $W(k,c,:,:)\odot I(c,:,:)$ with $I(c,:,:)$ of size $H_i \times W_i$ can be broken down into many convolutions each involving smaller tiles of I. This *overlapping* method can be illustrated by the following 1D example:

$W(0:2) \odot I(0:5) \longrightarrow O(0:5)$

$W(0:2) \odot \begin{bmatrix} I(0:3) \\ I(2:5) \end{bmatrix} \longrightarrow \begin{bmatrix} O(0:1) \\ O(2:3) \end{bmatrix}$

I is broken up into two tiles with some duplicating elements while O is partitioned (without duplication) into two tiles. More generally, given convolution kernels of size $r\times$ s and (small) sizes m, n that divide H_o and W_o, respectively, the input and output tensors can be reshaped: I, O into $\tilde{I}, \tilde{O}$

$I \in \mathbb{R}^{C\times H_i\times W_i} \longrightarrow \tilde{I} \in \mathbb{R}^{C\times T\times(m+r-1)\times(n+s-1)}$

and $O \in \mathbb{R}^{K\times H_o\times W_o} \longrightarrow \tilde{O} \in \mathbb{R}^{K\times T\times m\times n}$

Where T is the number of resulting tiles of the reshaping: $T = \frac{H_o W_o}{mn}$. The input

tile size is $(m+r-1)\times(n+s-1)$, while the output is $m\times n$. This reshaping can be expressed by two index mapping functions ϕ and ψ.
$\tilde{I}(c,t,i,j)=I(c,\phi(t,i,j)), O(k,i,j)=\tilde{O}(k,\psi(i,j))$
ϕ is many-to-one and maps a 3-tuple to a 2-tuple
ψ is invertible and maps a 2-tuple to a 3-tuple.
Using the overlapped form:

$$\tilde{O}(k,t,:,:)=\sum_{c=1}^{C} W(k,c,:,:)\odot\tilde{I}(c,t,:,:) \tag{3.17}$$

Solving equation3.17 in spatial domain takes $mnrs$ multiplications and $(m+r-1)(n+s-1)$ multiplications as shown in section 3.2. In Winograd domain, the output can be represented as:
$\tilde{O}(k,t,:,:)=A_1^T\left[\sum_{c=1}^{C}(W_f(k,c,:,:))\odot(B_1^T\tilde{I}(c,t,:,:)B_2)\right]A_2,$

$$\tilde{I}(c,t,i,j)=I(c,\phi(t,i,j)),\quad O(k,i,j)=\tilde{O}(k,\psi(i,j)) \tag{3.18}$$

For backward propagation, the partial derivatives of the scalar loss function L w.r.t. each of the variables $I(c,i,j)$ and $W_f(k,c,i,j)$ in terms of the known partial derivatives of L w.r.t. $O(k,i,j)$ $(=\frac{\partial L}{\partial O})$ has to be computed.

Suppose the partial derivatives of a scalar function L w.r.t. an array of variables y_{ij}, $Y\in\mathbb{R}^{\mu\times\nu}$, are known. Moreover, the variables Y are in fact dependent variables of an array of variables $x_{ij}, X\in\mathbb{R}^{\mu'\times\nu'}$ via $Y=U^TXV$ where U,V are constant matrices of commensurate dimensions. The partial derivatives of L w.r.t. X are then given by $\frac{\partial L}{\partial X}=U\frac{\partial L}{\partial Y}V^T$.

Let's denote the intermediate variables in Equation 3.18 by $\tilde{I}_f$ and Z : $\tilde{I}_f(c,t,:,:)=B_1^T\tilde{I}(c,t,:,:)B_2$ and $Z(k,t,:,:)=\sum_{c=1}^{C}W_f(k,c,:,:)\odot\tilde{I}_f(c,t,:,:)$ for all applicable indices. Here $Z(:,:,i,j)=W_f(:,:,i,j)\tilde{I}_f(:,:,i,j)$, which is the 2D slice of Z with any fixed i,j is a matrix product.
Now $\frac{\partial L}{\partial W_f}$ can be computed using chain rule as shown in equation 3.19:

$$\begin{aligned}\frac{\partial L}{\partial Z(k,t,:,:)}&=A_1\frac{\partial L}{\partial\tilde{O}(k,t,:,:)}A_2^T\\ \frac{\partial L}{\partial W_f(:,:,i,j)}&=\frac{\partial L}{\partial Z(:,:,i,j)}(\tilde{I}_f(:,:,i,j))^T\end{aligned} \tag{3.19}$$

here $\frac{\partial L}{\partial \tilde{O}}$ is $\frac{\partial L}{\partial O}$ with simple index mapping $\psi^{-}1$ [46].
Likewise, chain rule is applied to $\frac{\partial L}{\partial Z}$ computed in 3.19 to compute $\frac{\partial L}{\partial I}$:

$$\begin{aligned} \frac{\partial L}{\partial \tilde{I}_f(:,:,i,j)} &= (W_f(:,:,i,j))^T \frac{\partial L}{\partial Z(:,:,i,j)} \\ \frac{\partial L}{\partial \tilde{I}(c,t,:,:)} &= B_1 \frac{\partial L}{\partial \tilde{I}_f(c,t,:,:)} B_2^T \\ \frac{\partial L}{\partial I(c,i,j)} &= \sum_{(t,i',j');where(i,j)=\phi(t,i',j')} \frac{\partial L}{\partial \tilde{I}(c,t,i',j')} \end{aligned} \tag{3.20}$$

Equations 3.19 & 3.20 are analogous to partial derivatives in spatial domain(3.3 - 3.5). They are used to compute the gradient in Winograd domain and implement the backward pass of the network.

3.3 Implementation

Using the Winograd based convolution and gradient computation techniques explained in the previous sections(3.2.1 & 3.2.2), I started implementing the Winograd layer into TensorFlow[2]. The Winograd layer had to be implemented as a custom layer and integrated into the TensorFlow framework as a custom operation[3]. I had defined the custom operation but parts of my implementation were not working as expected. I could not find sufficient online documentation to be able to fix my implementation and had to switch to Caffe. I then implemented and integrated our Winograd layer into *Caffe* [50], which is a popular deep learning framework developed by Berkeley AI Research (BAIR)/The Berkeley Vision and Learning Center (BVLC) and community contributors. Our version of the Caffe framework with, Winograd layer integrated, is available in this[4] code repository.

A high-level detail related to Winograd layer and other optimization libraries used in the project is discussed in this section.

[2]TensorFlow is a free and open-source software library for data-flow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks.

[3]`https://www.tensorflow.org/guide/extend/op`

[4]`https://github.com/Swas99/caffe`

3.3.1 Implementation

In our experiments, the convolution operation is carried out between an input patch size of 4×4 and a kernel of size $k \times k$, where k $\in \{3, 5\}$. Kernel size of 3×3 and 5×5 were used in separate experiments. Using techniques mentioned in Winograd's seminal book [7], the Winograd parameters for F(4,3) and F(4,5) were computed. The constant parameters and all generic logic around Winograd's minimal filtering algorithm was incorporated in a utility class[5] and integrated into the Caffe framework.

To carry out the convolution in Winograd domain, we had to implement and integrate a new layer in the Caffe framework, that will replace the spatial convolution layer [6, 7, 8] . This layer receives the data(Blobs[9]) and passes on to the next layer after performing the convolution operation. During the back-propagation, the weights are adjusted according to the gradient *w.r.t* the loss function. The convolution operation is performed on image tiles and in mini-batches and arranging the memory block appropriately can help improve the performance. There are many popular techniques and libraries to perform this optimally [10, 11, 12] . We have also used them in our Winograd layer's implementation [13, 14].

3.3.2 Libraries

A high level detail of various libraries used in the project is discussed below:

- **SpMP:** SpMP(sparse matrix pre-processing) library includes optimized parallel implementations of a few key sparse matrix pre-processing routines: task dependency graph construction of Gauss-Seidel[15] like loops with data-dependent

[5]`https://github.com/Swas99/caffe/blob/master/include/caffe/util/winograd.hpp`
[6]`http://caffe.berkeleyvision.org/tutorial/layers/convolution.html`
[7]`https://github.com/Swas99/caffe/blob/master/src/caffe/layers/conv_layer.cpp`
[8]`https://github.com/Swas99/caffe/blob/master/src/caffe/layers/conv_layer.cu`
[9]`https://caffe.berkeleyvision.org/tutorial/net_layer_blob.html`
[10]`https://docs.nvidia.com/cuda/cublas/index.html`
[11]`https://stackoverflow.com/questions/46213531/how-is-using-im2col-operation-in-convolutional`
`47422548`
[12]`https://en.wikipedia.org/wiki/GEMM`
[13]`https://github.com/Swas99/caffe/blob/master/src/caffe/layers/winograd_layer.cpp`
[14]`https://github.com/Swas99/caffe/blob/master/src/caffe/layers/winograd_layer.cu`
[15]`https://en.wikipedia.org/wiki/GaussSeidel_method`

loop carried dependencies, and cache-locality optimizing re-orderings like breadth-first search (BFS) and reverse Cuthill-McKee[16] (RCM). In addition, SpMP includes auxiliary routines like parallel matrix transpose that is useful for moving back and forth between compressed sparse row (CSR) and compressed sparse column (CSC), matrix market file I/O, load balanced sparse matrix dense vector multiplication (SpMV), and optimized dissemination barrier.The pre-processing routines implemented in SpMP are very important for achieving high performance of key sparse matrix operations such as sparse triangular solver, Gauss-Seidel (GS) smoothing, incomplete LU(ILU) factorization, and SpMV, particularly in modern machines with many cores and deep memory hierarchy. It is also designed to showcase a "best known method" in high-performance parallel implementations of pre-processing routines.

- **Intel MKL:** Intel Math Kernel Library (Intel MKL) is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS[17], LAPACK[18], ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math.

- **cuBLAS:** The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA run-time. It allows the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU).

- **LMDB:** cublas Lightning Memory-Mapped Database (LMDB) is a software library that provides a high-performance embedded transactional database in the form of a key-value store. LMDB is written in C with API bindings for several programming languages. LMDB stores arbitrary key/data pairs as byte arrays, has a range-based search capability, supports multiple data items for a single key and has a special mode for appending records at the end of the database which gives a dramatic write performance increase over other similar stores. LMDB is not a relational database, it is strictly a key-value store.

[16]https://people.sc.fsu.edu/\protect\unhbox\voidb@x\hbox{~}jburkart/m_src/rcm/rcm.html

[17]https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[18]https://en.wikipedia.org/wiki/LAPACK

- **OpenCV:** OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.

- **cuSparse:** The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on top of the NVIDIA CUDA runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++.

- **Boost:** Boost is a set of libraries for the C++ programming language that provide support for tasks and structures such as multithreading, linear algebra, pseudo-random number generation, regular expressions, image processing, and unit testing.

- **OpenMP:** OpenMP is a library for parallel programming in the SMP (symmetric multi-processors, or shared-memory processors) model. When programming with OpenMP, all threads share memory and data.

- **cuDNN:** The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. Deep learning researchers and framework developers worldwide rely on cuDNN for high-performance GPU acceleration. It allows them to focus on training neural networks and developing software applications rather than spending time on low-level GPU performance tuning. cuDNN accelerates widely used deep learning frameworks.

- **HDF5:** HDF5 is a unique technology suite that makes possible the management of extremely large and complex data collections. The HDF5 technology suite includes: A versatile data model that can represent very complex data objects and a wide variety of metadata.

Environment Details		
Name	**Version**	**Description/Type**
Caffe	0.17.3	Deep learning framework
OpenCV	4.1.1	Vision Library
Boost	1.70	C++ library
LMDB	0.9.24	Lightning Memory-Mapped Database

Table 3.1: Environment Details

Environment Details		
Name	**Version**	**Description/Type**
Intel MKL	2019.4.243	Math Kernel Library
OpenMP	19.0.4	parallel programming library
LMDB	0.9.24	Lightning Memory-Mapped Database
SpMP	0.7	Sparse matrix pre-processing
Cuda	10.1	Parallel computing platform & programming model for GPU
cuDNN	7.6.1.34	CUDA Deep Neural Network
GPU	TU106 [GeForce RTX 2070]	GPU
CPU	AMD Ryzen 5 2600X Six-Core Processor	CPU; x86_64
Arch Linux	Linux 5.2.6-arch1-1-ARCH	OS

Table 3.2: Environment Details

3.3.3 Experiment Environment

Specific version of the software and hardware used in this research is described in table 3.1 & 3.2.

Chapter 4

Experiments

Using our Winograd layer, we have conducted different experiments to capture different metrics and compare the results. The experiments conducted can be broadly classified as mentioned in table 4.1.

Experiment description					
Sl.No.	Data-set	Mode	Domain	Kernel size	Net
1	MNIST	CPU	Spatial	3×3	LeNet
2	MNIST	CPU	Winograd	3×3	LeNet
3	MNIST	CPU	Spatial	5×5	LeNet
4	MNIST	CPU	Winograd	5×5	LeNet
5	MNIST	GPU	Spatial	3×3	LeNet
6	MNIST	GPU	Winograd	3×3	LeNet
7	MNIST	GPU	Spatial	5×5	LeNet
8	MNIST	GPU	Winograd	5×5	LeNet
9	CIFAR-10	CPU	Spatial	3×3	VGG-19
10	CIFAR-10	CPU	Winograd	3×3	VGG-19
11	CIFAR-10	CPU	Spatial	5×5	VGG-19
12	CIFAR-10	CPU	Winograd	5×5	VGG-19
13	CIFAR-10	GPU	Spatial	3×3	VGG-19
14	CIFAR-10	GPU	Winograd	3×3	VGG-19
15	CIFAR-10	GPU	Spatial	5×5	VGG-19
16	CIFAR-10	GPU	Winograd	5×5	VGG-19

Table 4.1: Experiment description

Key metrics captured for each experiment are:
1. Accuracy of the trained model;

2. Execution time per 100 iterations of training; and
3. Total training time to reach acceptable accuracy.

The trained weights were then pruned in an iterative manner and it's effect on model's accuracy was studied.

4.1 Data set

A brief description of the data sets used in the experiments is provided in the table 4.2.

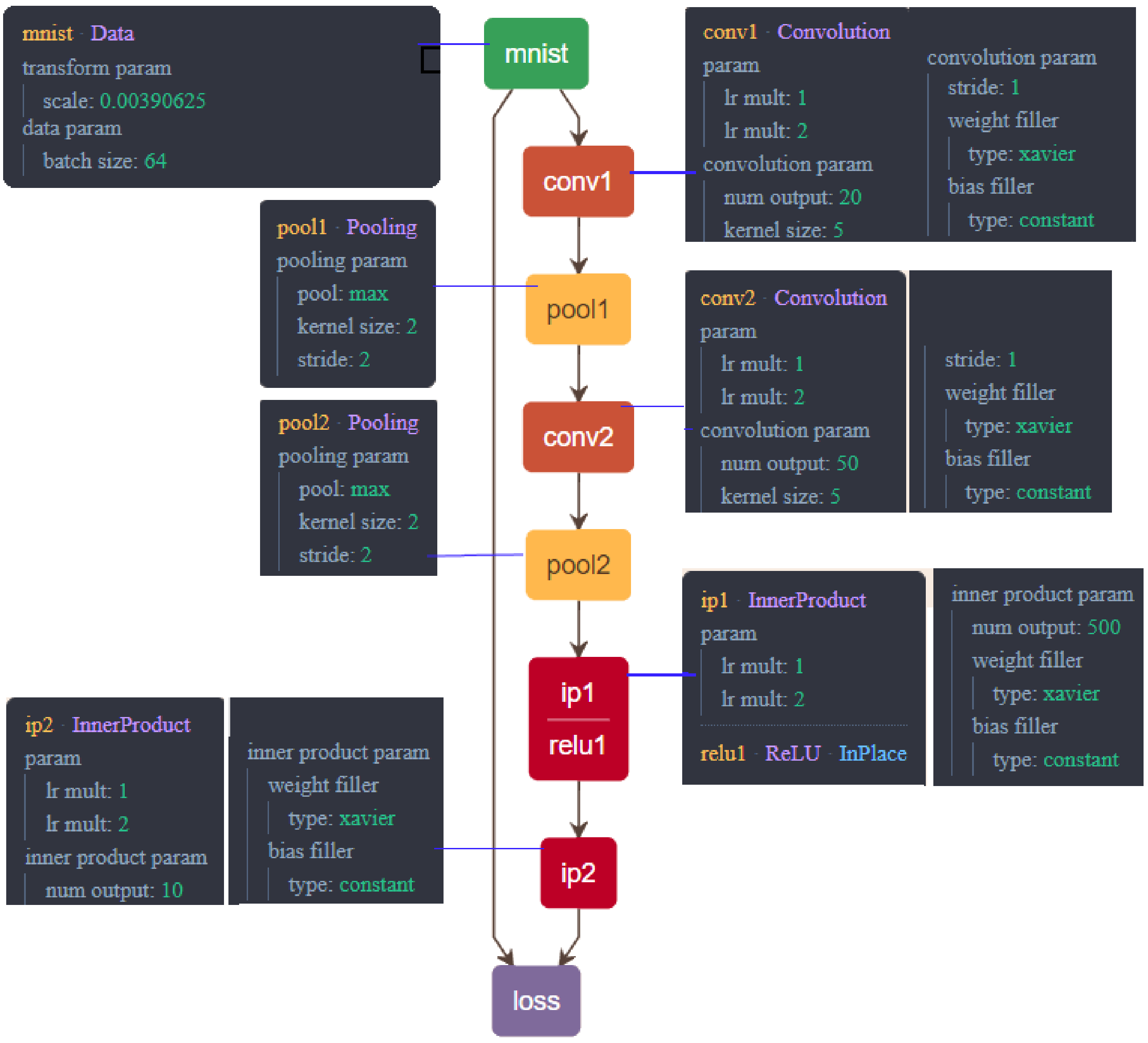
Data set description				
Data-set	Description	Size	Channels	Contents
MNIST	large database of handwritten digits	28x28 pixel	1(gray-scale)	60,000 training images and 10,000 testing images
CIFAR-10	colour images in 10 classes, with 6000 images per class	32x32 pixel	3(RGB)	50,000 training images and 10,000 testing images

Table 4.2: Data set description

4.2 CNN Architecture

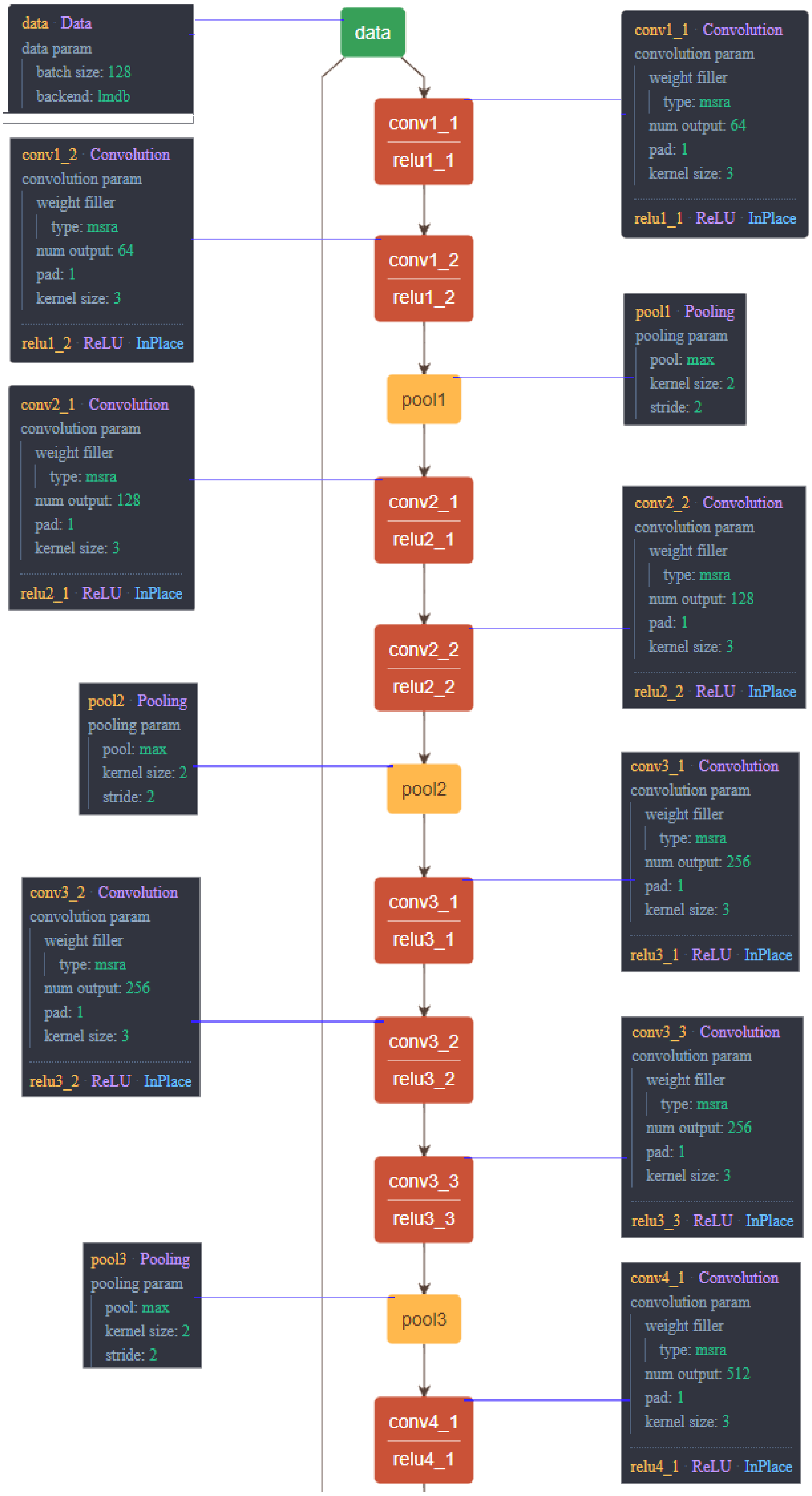
For all the experiments with MNIST data set, LeNet architecture was used. The convolution layer(Spatial/Winograd) and kernel size(3×3 or 5×5) was set as per the experiment. Full details of the architecture is illustrated in figure 4.1.

For all the experiments with CIFAR-10 data set, a VGG architecture with 19 layers was used. The convolution layer(Spatial/Winograd) and kernel size(3×3 or 5×5) was set as per the experiment. The architecture is illustrated in figures 4.2 & 4.3.



```
mnist · Data
transform param
  scale: 0.00390625
data param
  batch size: 64

conv1 · Convolution
param
  lr mult: 1
  lr mult: 2
convolution param
  num output: 20
  kernel size: 5
convolution param
  stride: 1
  weight filler
    type: xavier
  bias filler
    type: constant

pool1 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2

conv2 · Convolution
param
  lr mult: 1
  lr mult: 2
convolution param
  num output: 50
  kernel size: 5
  stride: 1
  weight filler
    type: xavier
  bias filler
    type: constant

pool2 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2

ip1 · InnerProduct
param
  lr mult: 1
  lr mult: 2
relu1 · ReLU · InPlace
inner product param
  num output: 500
  weight filler
    type: xavier
  bias filler
    type: constant

ip2 · InnerProduct
param
  lr mult: 1
  lr mult: 2
inner product param
  num output: 10
inner product param
  weight filler
    type: xavier
  bias filler
    type: constant
```

Figure 4.1: LeNet



```
data · Data
data param
  batch size: 128
  backend: lmdb
```

```
conv1_1 · Convolution
convolution param
  weight filler
    type: msra
  num output: 64
  pad: 1
  kernel size: 3
relu1_1 · ReLU · InPlace
```

```
conv1_2 · Convolution
convolution param
  weight filler
    type: msra
  num output: 64
  pad: 1
  kernel size: 3
relu1_2 · ReLU · InPlace
```

```
pool1 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2
```

```
conv2_1 · Convolution
convolution param
  weight filler
    type: msra
  num output: 128
  pad: 1
  kernel size: 3
relu2_1 · ReLU · InPlace
```

```
conv2_2 · Convolution
convolution param
  weight filler
    type: msra
  num output: 128
  pad: 1
  kernel size: 3
relu2_2 · ReLU · InPlace
```

```
pool2 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2
```

```
conv3_1 · Convolution
convolution param
  weight filler
    type: msra
  num output: 256
  pad: 1
  kernel size: 3
relu3_1 · ReLU · InPlace
```

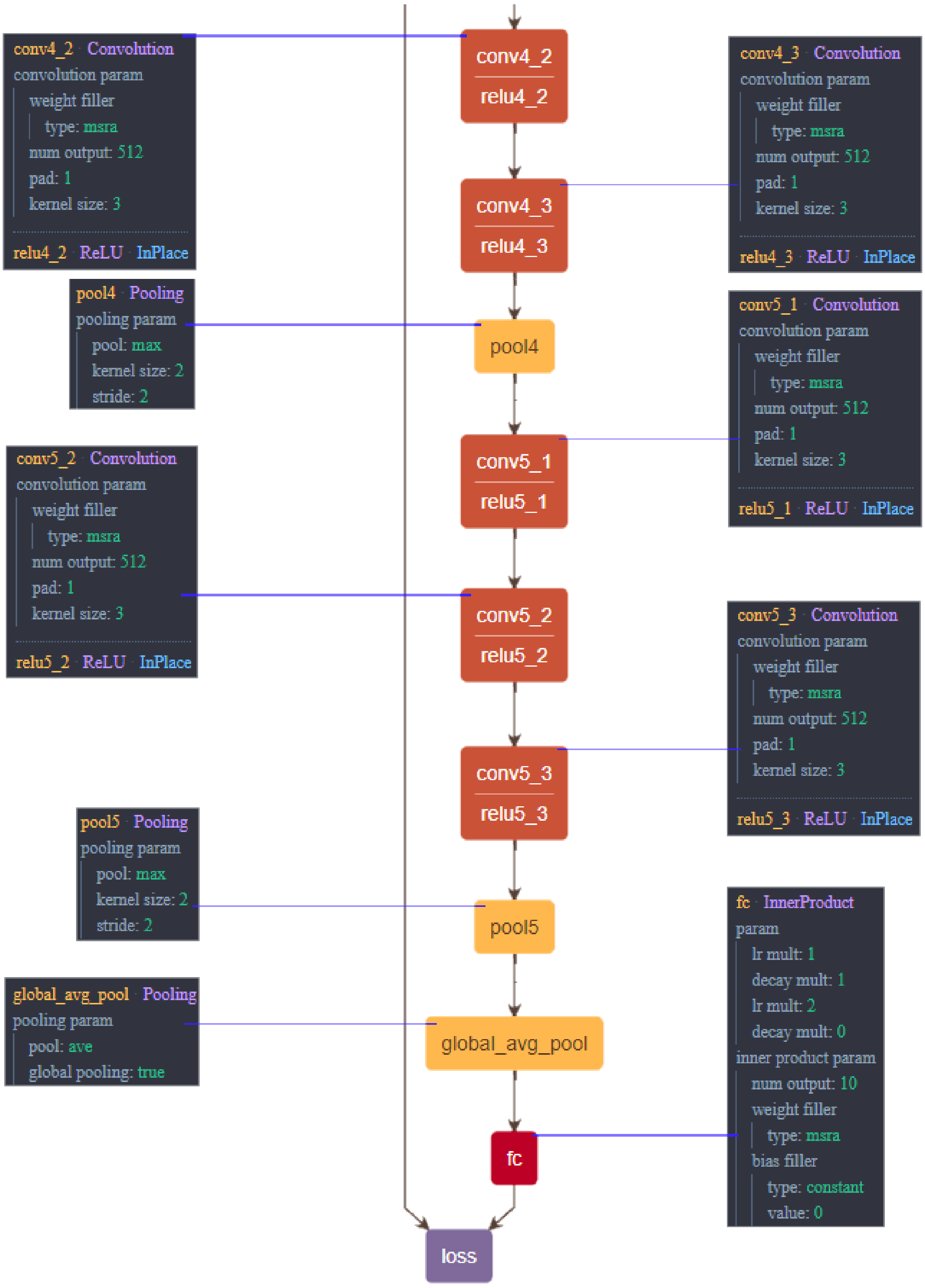
```
conv3_2 · Convolution
convolution param
  weight filler
    type: msra
  num output: 256
  pad: 1
  kernel size: 3
relu3_2 · ReLU · InPlace
```

```
conv3_3 · Convolution
convolution param
  weight filler
    type: msra
  num output: 256
  pad: 1
  kernel size: 3
relu3_3 · ReLU · InPlace
```

```
pool3 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2
```

```
conv4_1 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu4_1 · ReLU · InPlace
```

Figure 4.2: VGG Net



```
conv4_2 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu4_2 · ReLU · InPlace

conv4_3 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu4_3 · ReLU · InPlace

pool4 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2

conv5_1 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu5_1 · ReLU · InPlace

conv5_2 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu5_2 · ReLU · InPlace

conv5_3 · Convolution
convolution param
  weight filler
    type: msra
  num output: 512
  pad: 1
  kernel size: 3
relu5_3 · ReLU · InPlace

pool5 · Pooling
pooling param
  pool: max
  kernel size: 2
  stride: 2

global_avg_pool · Pooling
pooling param
  pool: ave
  global pooling: true

fc · InnerProduct
param
  lr mult: 1
  decay mult: 1
  lr mult: 2
  decay mult: 0
inner product param
  num output: 10
  weight filler
    type: msra
  bias filler
    type: constant
    value: 0
```

Figure 4.3: VGG Net

4.3 Hyper-Parameters

The hyper-parameters used in our experiment on MNIST data set with LeNet are presented in table 4.3, and that for our experiments on CIFAR-10 data set with VGG net are presented in table 4.4
The brief description of parameters used in Caffe's solver file is present in this link[1].

LeNet hyper-parameters
base_lr: 0.000001
lr_policy: "inv"
momentum: 0.9
weight_decay: 0.00005
gamma: 0.0001
power: 0.75

Table 4.3: LeNet hyper-parameters

VGG Net hyper-parameters
base_lr: 0.1
lr_policy: "multistep"
momentum: 0.9
weight_decay: 0.0001
gamma: 0.1
stepvalue: 32000
stepvalue: 48000
type: "Nesterov"

Table 4.4: VGG Net hyper-parameters

4.4 Results

4.4.1 Training experiments

Our GPU implementation of training in Winograd domain was more than $\sim 2\times$ times faster than training in spatial domain across all experiments. However, our CPU implementation of training in Winograd domain was slower. Table 4.5 shows the average time per 100 iterations across all experiments.

For our experiments with MNIST data-set, the Winograd convolution converged much quicker than the spatial convolution. For GPU training with 3×3 kernel, the Winograd convolution model has reached a top-5 accuracy score of ~85% in 1000 iterations and ~94% in 10000 iterations. Whereas the spatial convolution model on GPU training with a 3×3 kernel has a top-5 accuracy score of ~13% in 1000 iterations and ~29% in 10000 iterations. For CPU training with 3×3 kernel, the Winograd convolution model has reached a top-5 accuracy score of ~86% in 300 iterations and

[1]`https://github.com/BVLC/caffe/wiki/Solver-Prototxt`

Execution time					
Data	**Net**	**Mode**	**Domain**	**Kernel size**	**Average execution time per 100 iterations**
MNIST	LeNet	GPU	Spatial	3×3	1365 ms
MNIST	LeNet	GPU	Winograd	3×3	701 ms
MNIST	LeNet	CPU	Spatial	3×3	26573 ms
MNIST	LeNet	CPU	Winograd	3×3	43190 ms
MNIST	LeNet	GPU	Spatial	5×5	1653 ms
MNIST	LeNet	GPU	Winograd	5×5	669 ms
MNIST	LeNet	CPU	Spatial	5×5	25898 ms
MNIST	LeNet	CPU	Winograd	5×5	41409 ms
CIFAR-10	VGG19	GPU	Spatial	3×3	44080 ms
CIFAR-10	VGG19	GPU	Winograd	3×3	12890 ms
CIFAR-10	VGG19	CPU	Spatial	3×3	709723 ms
CIFAR-10	VGG19	CPU	Winograd	3×3	818265 ms
CIFAR-10	VGG19	GPU	Spatial	5×5	43125 ms
CIFAR-10	VGG19	GPU	Winograd	5×5	12345 ms
CIFAR-10	VGG19	CPU	Spatial	5×5	708423 ms
CIFAR-10	VGG19	CPU	Winograd	5×5	817541 ms

Table 4.5: Execution time

~94% in 2500 iterations. The spatial convolution model on CPU training with 3×3 kernel has a top-5 accuracy score of ~32% in 300 iterations and ~78% in 2500 iterations. A similar trend is observed when training with a 5×5 kernel. Top-5 accuracy scores of ~87% & ~94% are observed at 300^{th} & 4500^{th} iteration respectively for Winograd convolution, and that of ~10% & ~22% are observed at 300^{th} & 4500^{th} iteration respectively for spatial convolution, when training on GPU. For CPU training, top-5 accuracy scores of ~86% & ~94% are observed at 400^{th} & 4500^{th} iteration respectively for Winograd convolution, and that of ~58% & ~ 94% are observed at 400^{th} & 4500^{th} iteration respectively for spatial convolution.

The results of training with MNIST with different configurations is shown as accuracy vs iteration graph in figures 4.4 & 4.5.

Training on CIFAR-10 was not straight forward and we had to try different combination of hyper-parameters and network architectures. Using the hyper-parameters

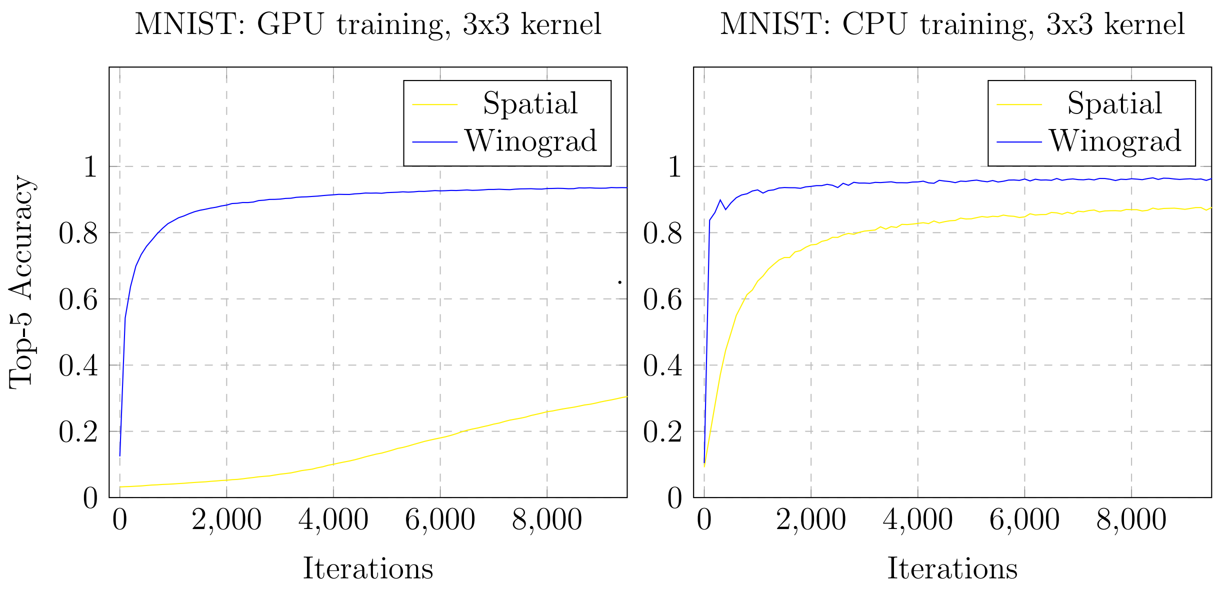


Figure 4.4: MNIST training with 3x3 kernel

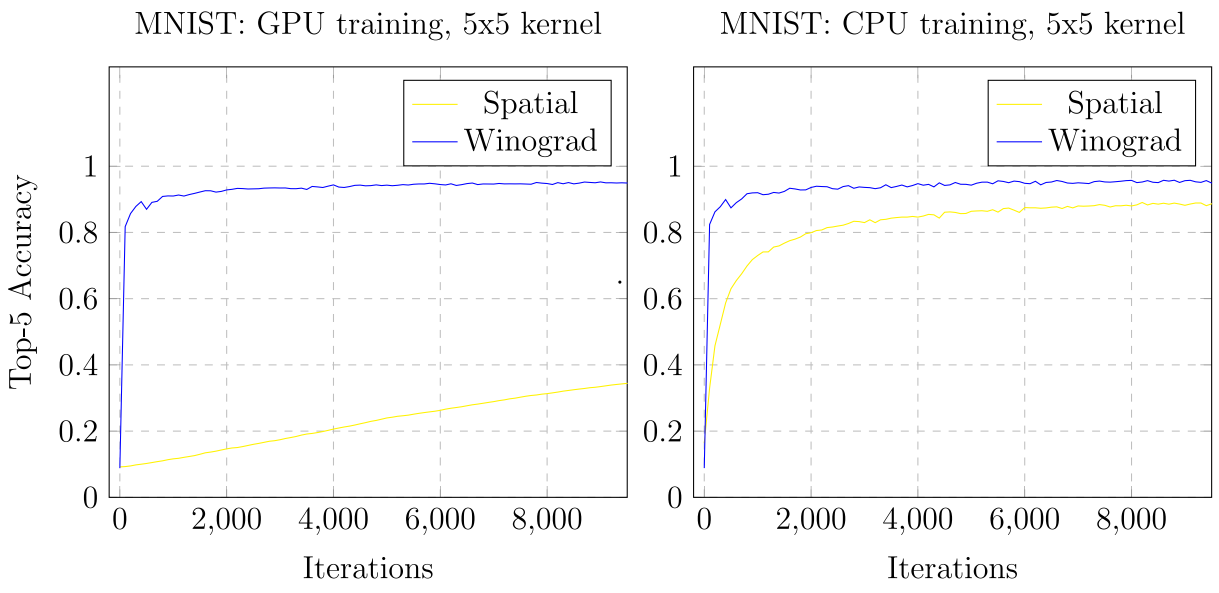


Figure 4.5: MNIST training with 5x5 kernel

and architecture discussed in section 4.2 & 4.3, we were able to train a successful model using our GPU implementation. We are not able to train a successful model using our CPU implementation. The spatial convolution-based model was observed to stabilize quicker than the Winograd convolution-based model. In terms of number of iterations, the spatial based convolution had converged earlier than the Winograd based model but the effective time taken for convergence was better for the Winograd based model.

The results of training with CIFAR-10 with different configurations is shown as accuracy vs iteration graph in figures 4.6 & 4.7.

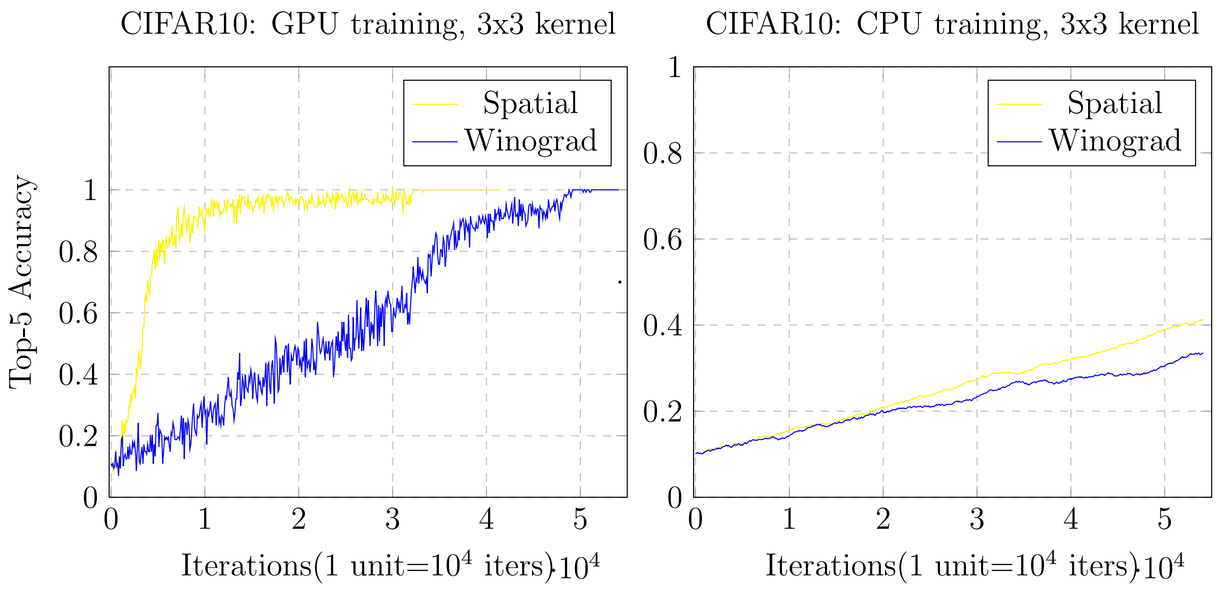


Figure 4.6: CIFAR10 training with 3x3 kernel

4.4.2 Pruning experiments

The trained weights were pruned in an iterative manner, by discarding the lowest weights in each iteration, and the accuracy with the pruned weight was noted. The results of the pruning experiments are shown as accuracy vs sparsity graph in figures 4.8 - 4.11. The rate of accuracy drop per pruned weight was observed to be higher for weights trained in spatial domain.

The results for training and pruning experiments are also available as a CSV file in this[2] link.

[2]`https://github.com/Swas99/caffe/tree/master/results`

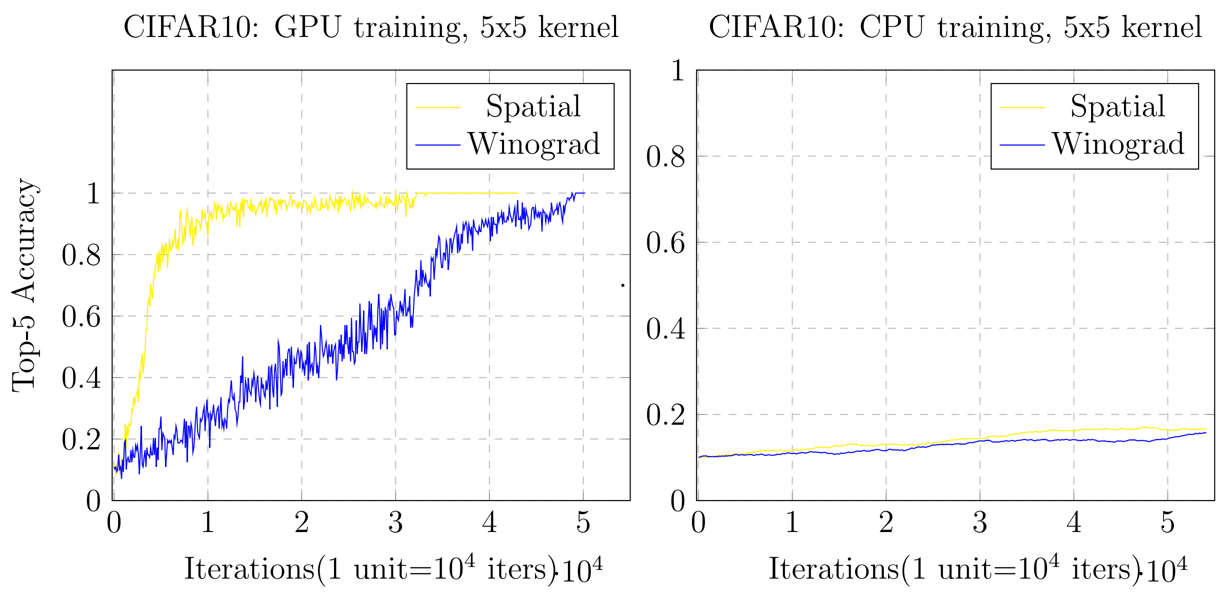


Figure 4.7: CIFAR10 training with 5x5 kernel

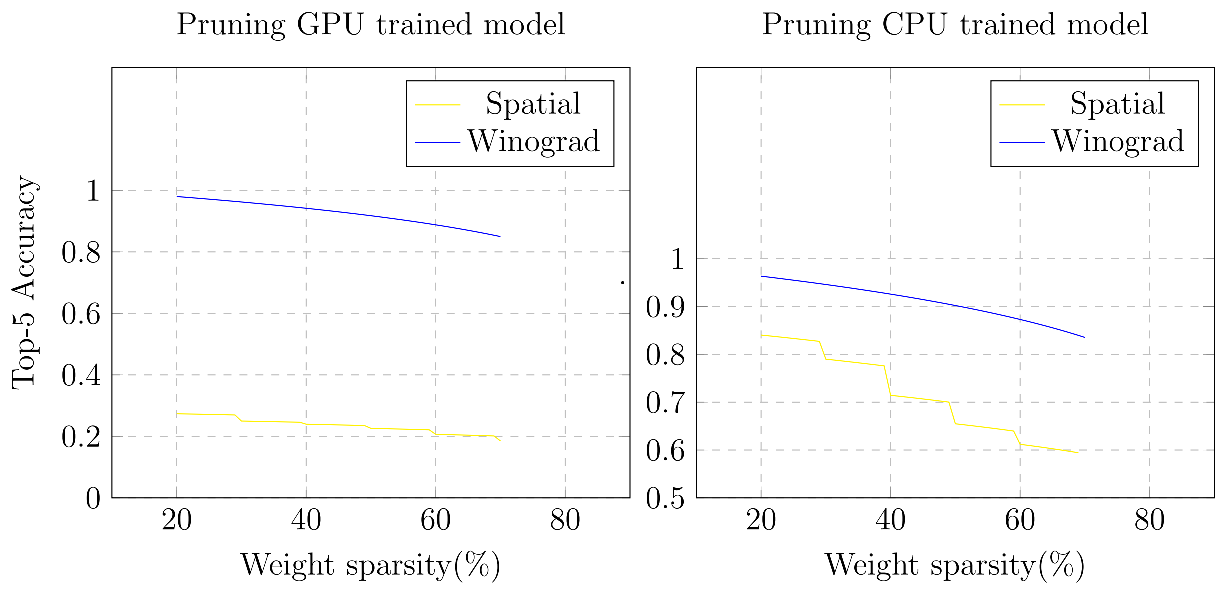


Figure 4.8: MNIST: Pruning models trained with 3x3 kernel

4.5 Discussion

For experiments with MNIST: better accuracy levels were observed for Winograd convolution-based models across all experiments. The GPU training of Winograd

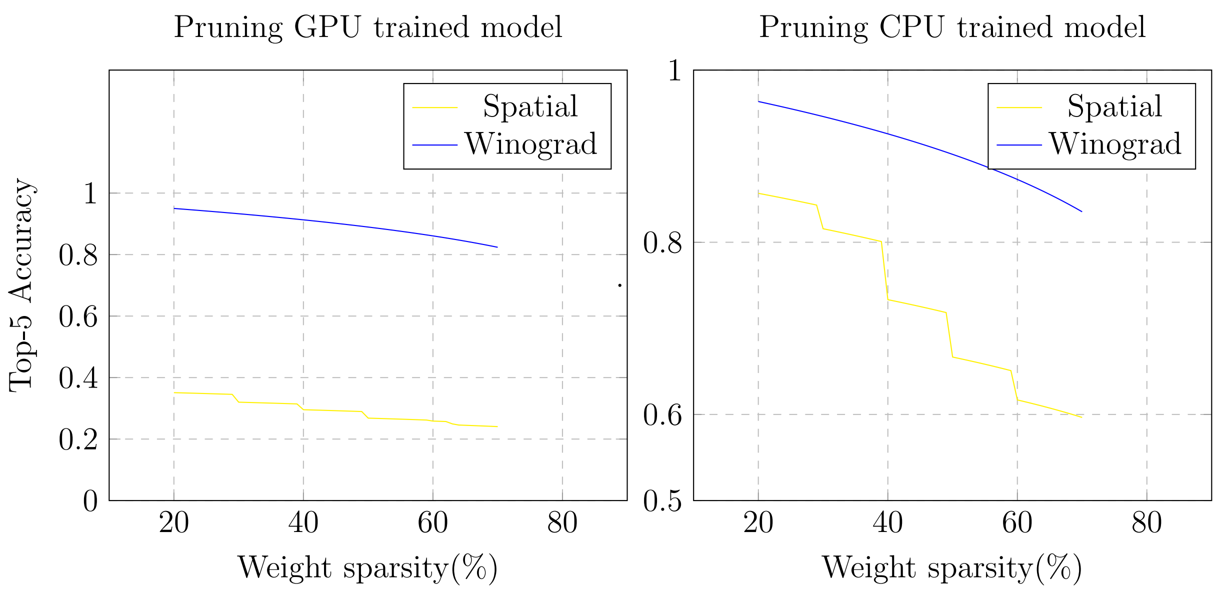


Figure 4.9: MNIST: Pruning models trained with 5x5 kernel

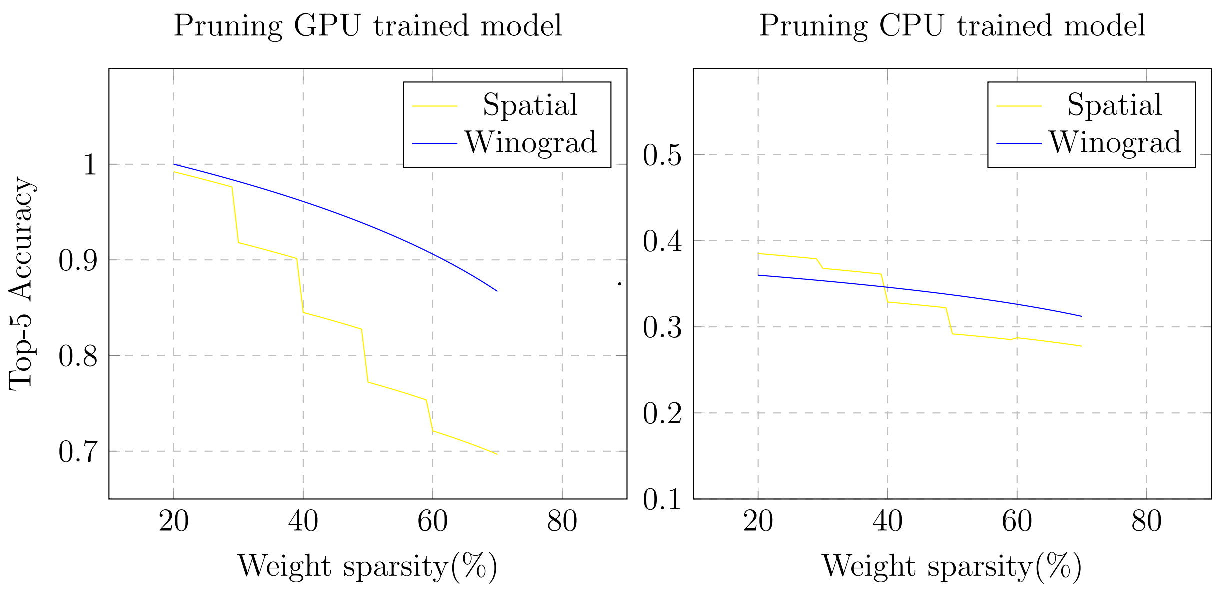


Figure 4.10: CIFAR10: Pruning models trained with 3x3 kernel

convolution-based models had reached accuracy levels of ∼90% within 20 seconds of training and that of ∼94% within 70 seconds of training time. The accuracy levels achieved by the spatial model on GPU training after 20 seconds and 70 seconds of training were ∼13% and ∼20% respectively. Higher accuracy levels were observed for

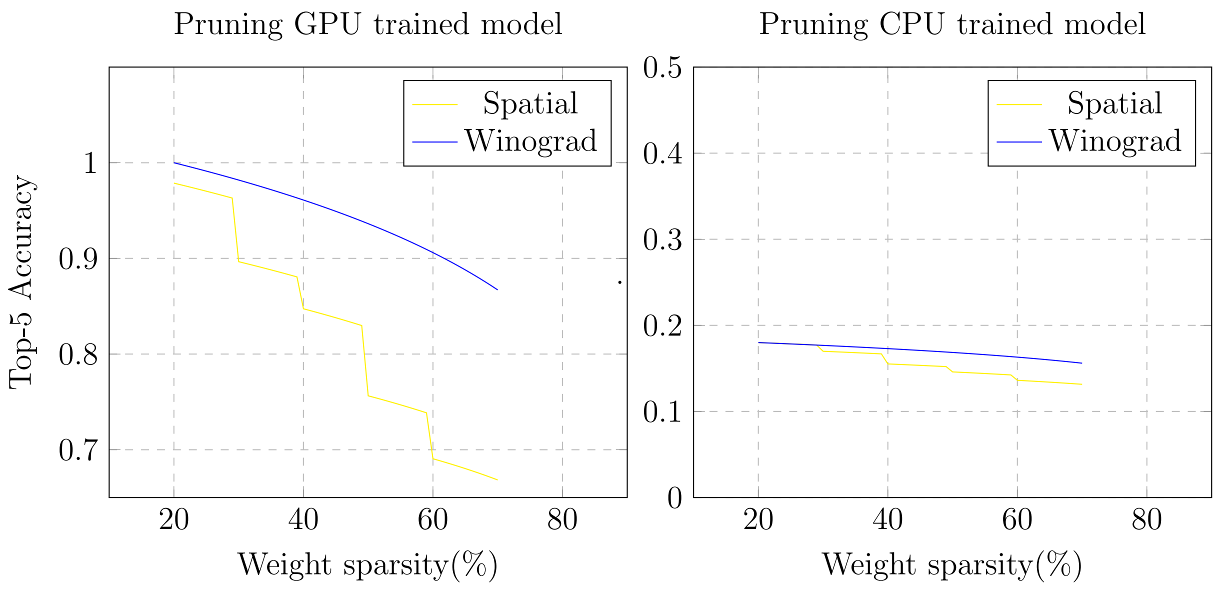


Figure 4.11: CIFAR10: Pruning models trained with 5x5 kernel

all our training with MNIST data-set on Winograd domain compared to that on spatial domain.

For experiments with CIFAR-10: The GPU training of Winograd convolution-based models had stabilized around ∼47000 iterations, with an accuracy of ∼92%, and the total time elapsed at this point was ∼1h 38m. The spatial convolution-based model on GPU training had stabilized around ∼30000 iterations, with an accuracy of ∼95%, and the total time elapsed at this point was ∼3h 40m. The Winograd based model took more iterations to converge but the effective training time was more than 2× faster.

Overall, training CNNs in Winograd domain is much faster than training them in spatial domain. The speedups observed are higher on larger data-sets, where the performance gain is significantly higher than the overhead involved in Winograd convolution. By fine-tuning the model appropriately similar (or better) accuracy levels can also be achieved for models trained in Winograd domain.

4.6 Limitations

Due to time constraint we could not fine-tune the CIFAR-10 model for our experiments on CPU. We were unable to successfully train our model to reach acceptable accuracy levels, and the execution time per iterations in Winograd domain was also much higher than that in spatial domain. Park et al. [46] were able to achieve speedups of more than 2× with minimum accuracy loss for their Winograd convolution-based model using CPU and thread-level optimization.

Experiments with ImageNet, which is a huge database of 224×224 colour images with 14 Million+ images in 20,000+ categories, can take over 10-15 days of training time. It was out of scope to perform any experiments with the ImageNet data-set.

4.7 Future Work

Winograd convolution with F(4,3) and F(4,5) were used in this experiment. Larger patch sizes can be used and the effect on training time and accuracy can be studied. A simple approach to prune the trained weights was used in this research. Sophisticated pruning techniques [33, 38, 51], some of which may be incorporated during training, can be explored. Both training time and accuracy is expected to improve using sophisticated pruning algorithms. Lastly, it would to interesting to conduct experiments on the ImageNet data-set with the existing implementation.

Chapter 5

Conclusion

Our GPU implementation of Winograd convolution is $\sim 2\times$ times faster than convolution in spatial domain for MNIST data-set, and $\sim 3.4\times$ times faster for CIFAR-10 data-set. Higher accuracy levels and quicker convergence was observed when training in Winograd domain on the MNIST data-set. For CIFAR-10 data-set, the effective time to converge when training in Winograd domain is $\sim 2.25\times$ times faster compared to training in spatial domain. The application of pruning methods to models trained in Winograd domain performs better than the models trained in the spatial domain. By applying processor-level optimization techniques, the Winograd based models are expected to outperform the spatial domain models even on CPUs.

Training deep networks with large data-sets can be made feasible using Winograd based techniques. Fast inference can enable use in critical systems like autonomous vehicles. Low power IoT devices which lack computational resources do not have to depend on a central server to process their data. One use case would be to deploy training capability on devices which can sustain them. These devices can process the data in real-time to build their own predictive model and work offline if necessary. A simple example could be, IoT sensors that capture light/noise levels and build a predictive model at the edge, specific to the deployed location of the device, and may trigger some action according to the input data. Another simple example could be, deploying the training capability on mobile phones that consists of hundreds (if not thousands) of images. Using Winograd based techniques a large number of images can be processed on a users device. The machine learning model may be used to cluster images into groups or perform some other task to provide a better user experience. In all of these scenarios, the bottleneck of transferring and handling huge amounts of data can be removed. The edge/mobile device can work independently. Even certain privacy issues can be solved. Not everyone is willing to share the images that reside in their mobile devices with a third party server and if we can process the image on their device itself, then there is no need to. In a different scenario, where it is not possible to deploy a model that can perform the training phase, a pre-trained model can be deployed. The inference latency for Winograd convolution-based models is at least $2\times$ faster than that for spatial convolution-based models across all of our experiments. Winograd convolution-based models are also light-weight in terms of memory and other computational requirements. Combining other optimization techniques like pruning, model compression can help increase the efficiency even further.

Winograd based techniques can help immensely in optimizing CNN's performance. It can be applied to a variety of applications from wearable(s), autonomous vehicles, virtual assistants to health-care. The ever increasing workload on the cloud infrastructure can be reduced significantly by using these techniques.

Bibliography

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *arXiv e-prints*, p. arXiv:1703.09039, Mar 2017.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, November 1998.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[7] S. Winograd, "Arithmetic complexity of computations," *Siam*, vol. volume 33, 1980.

[8] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4013–4021, 2015.

[9] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A gpu performance evaluation," *CoRR*, vol. abs/1412.7580, 2014.

[10] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *CoRR*, vol. abs/1312.5851, 2013.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, Winter 1989.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Intelligent Signal Processing*, pp. 306–351, IEEE Press, 2001.

[14] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, Apr 1980.

[15] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," *Tenth International Workshop on Frontiers in Handwriting Recognition*, 10 2006.

[16] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pp. 1237–1242, AAAI Press, 2011.

[17] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, pp. 354–356, Aug. 1969.

[18] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks,” in *Artificial Neural Networks and Machine Learning – ICANN 2014* (S. Wermter, C. Weber, W. Duch, T. Honkela, P. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, eds.), (Cham), pp. 281–290, Springer International Publishing, 2014.

[19] D. H. B. C. S. Heideman, Michael T.; Johnson, “Gauss and the history of the fast fourier transform,” *IEEE ASSP Magazine*, vol. doi:10.1109/MASSP.1984.1162257, pp. 1–9, 1984.

[20] C. Van Loan, “Computational frameworks for the fast fourier transform,” *SIAM*, 1992.

[21] D. Coppersmith and S. Winograd, “Matrix multiplication via arithmetic progressions,” *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251 – 280, 1990. Computational algebraic complexity editorial.

[22] A. Zlateski, Z. Jia, K. Li, and F. Durand, “Fft convolutions are faster than winograd on modern cpus, here is why,” *ArXiv*, vol. abs/1809.07851, 2018.

[23] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *ArXiv*, vol. abs/1806.08342, 2018.

[24] L. Lai, N. Suda, and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” *ArXiv*, vol. abs/1801.06601, 2018.

[25] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, 2016.

[26] L. Meng and J. Brothers, “Efficient winograd convolution via integer arithmetic,” *ArXiv*, vol. abs/1901.01965, 2019.

[27] P. Maji, A. Mundy, G. Dasika, J. G. Beu, M. Mattina, and R. Mullins, “Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus,” *ArXiv*, vol. abs/1903.01521, 2019.

[28] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size," *ArXiv*, vol. abs/1602.07360, 2017.

[29] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2015.

[30] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems 5* (S. J. Hanson, J. D. Cowan, and C. L. Giles, eds.), pp. 164–171, Morgan-Kaufmann, 1993.

[31] S. W. Stepniewski and A. J. Keane, "Pruning backpropagation neural networks using modern stochastic optimisation techniques," *Neural Computing & Applications*, vol. 5, pp. 76–98, Jun 1997.

[32] 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 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 1135–1143, Curran Associates, Inc., 2015.

[33] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[34] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *NIPS*, 2015.

[35] G. Castellano, A. M. Fanelli, and M. Pelillo, "An iterative pruning algorithm for feedforward neural networks," *IEEE Transactions on Neural Networks*, vol. 8, pp. 519–531, May 1997.

[36] M. D. Collins and P. Kohli, "Memory bounded deep convolutional networks," *ArXiv*, vol. abs/1412.1442, 2014.

[37] A. Polyak and L. Wolf, "Channel-level acceleration of deep face representations," *IEEE Access*, vol. 3, pp. 2163–2175, 2015.

[38] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *JETC*, vol. 13, pp. 32:1–32:18, 2015.

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[40] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1058–1066, PMLR, 17–19 Jun 2013.

[41] C. Louizos, M. Welling, and D. P. Kingma, "Learning Sparse Neural Networks through L_0 Regularization," *arXiv e-prints*, p. arXiv:1712.01312, Dec 2017.

[42] Y. Gal, J. Hron, and A. Kendall, "Concrete Dropout," *arXiv e-prints*, p. arXiv:1705.07832, May 2017.

[43] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational Dropout Sparsifies Deep Neural Networks," *arXiv e-prints*, p. arXiv:1701.05369, Jan 2017.

[44] D. P. Kingma, T. Salimans, and M. Welling, "Variational dropout and the local reparameterization trick," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 2575–2583, Curran Associates, Inc., 2015.

[45] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *ArXiv*, vol. abs/1802.06367, 2017.

[46] S. R. Li, J. Park, and P. T. P. Tang, "Enabling sparse winograd convolution by native pruning," *ArXiv*, vol. abs/1702.08597, 2017.

[47] X. Liu and Y. Turakhia, "Pruning of winograd and fft based convolution algorithm," in *Stanford edu*, 2016.

[48] Y. Huan, Y. Qin, Y. You, L. Zheng, and Z. Zou, "A multiplication reduction technique with near-zero approximation for embedded learning in iot devices,"

2016 29th IEEE International System-on-Chip Conference (SOCC), pp. 102–107, 2016.

[49] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 1–13, IEEE Press, 2016.

[50] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[51] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *ICLR*, 2016.