BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

NEURAL NETWORK IMPLEMENTATION WITHOUT MULTIPLICATION

IMPLEMENTACE NEURONOVÉ SÍTĚ BEZ OPERACE NÁSOBENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **BC. LUKÁŠ SLOUKA**
AUTOR PRÁCE

SUPERVISOR **Ing. IGOR SZŐKE, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání diplomové práce/19117/2017/xslouk02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Slouka Lukáš, Bc.**
Obor: Matematické metody v informačních technologiích
Téma: **Implementace neuronové sítě bez operace násobení**
Neural Network Implementation without Multiplication
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s teorií umělých neuronových sítí. Seznamte se s nástroji CNTK, TensorFlow, Kaldi nebo Theano.
2. Natrénujte jednoduchou dopřednou neuronovou síť standardní cestou.
3. Upravte nástroj a trénovací postup tak, aby došlo k odstranění operace násobení.
4. Natrénujte síť bez použití operace násobení. Porovnejte se standardní implementací.
5. Analyzujte úspěšnost binarizačních technik na různých typech neuronových sítí z pohledu přesnosti.
6. Zhodnoťte dosažené výsledky a navrhněte směry dalšího vývoje.
7. Vytvořte A2 plakátek a cca 30 vteřinové video prezentující výsledky vašeho projektu.

Literatura:

- Z. Lin, M. Courbariaux, R. Memisevic, Y. Bengio, Neural Networks with Few Multiplications, ICLR 2016 (arxiv.org/abs/1510.03009)
- Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, Ali Farhadi, XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks (https://arxiv.org/abs/1603.05279)
- Dále dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 ze zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Szőke Igor, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

The subject of this thesis is neural network acceleration with the goal of reducing the number of floating point multiplications. The theoretical part of the thesis surveys current trends and methods used in the field of neural network acceleration. However, the focus is on the binarization techniques which allow replacing multiplications with logical operators. The theoretical base is put into practice in two ways. First is the GPU implementation of crucial binary operators in the TensorFlow framework with a performance benchmark. Second is an application of these operators in simple image classifier. Results are certainly encouraging. Implemented operators achieve speed-up by a factor of 2.5 when compared to highly optimized cuBLAS operators. The last chapter compares accuracies achieved by binarized models and their full-precision counterparts on various architectures.

Abstrakt

Predmetom tejto diplomovej práce je akcelerácia neurónových sietí s cieľom redukcie počtu operácií násobenia reálnych čísiel. Teoretická časť tejto práce sleduje súčasné trendy a metódy využívané v oblasti akcelerácie neurónových sietí. Najväčší dôraz je kladený na binarizačné techniky, ktoré umožňujú nahradiť násobenia logickými operátormi. Teoretický základ je zavedený do praxe hneď dvomi spôsobmi. Prvým z nich je implementácia kritických binárnych operátorov spustiteľných na GPU vo frameworku TensorFlow a ich rýchlostný benchmark. Druhým je aplikácia týchto operátorov v jednoduchom klasifikátore obrázkov. Výsledky sú rozhodne povzbudivé. Implementované operátory dosiahli 2,5-násobné zrýchlenie v porovnaní s vysoko optimalizovanými cuBLAS operátormi. Posledná kapitola práce sleduje úspešnosť dosiahnutú binarizačnými modelmi.

Keywords

feedforward network, convolutional neural network, binarization, quantization, neural network acceleration, XNOR-network, BinaryConnect, BinaryNet, AlexNet, ImageNet, MNIST, CIFAR-10, SVHN, TensorFlow, matrix multiplication, CUDA, GPU

Kľúčové slová

dopredná neurónová sieť, konvolučná neurónová sieť, binarizácia, kvantizácia, akcelerácia neurónovej siete, XNOR-sieť, BinaryConnect, BinaryNet, AlexNet, ImageNet, MNIST, CIFAR-10, SVHN, TensorFlow, násobenie matíc, CUDA, GPU

Reference

SLOUKA, Lukáš. *Neural Network Implementation without Multiplication*. Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Igor Szőke, Ph.D.

Rozšírený abstrakt

Počas posledných niekoľko rokov sa z neurónových sietí stali najsofistikovanejšie prostriedky na riešenie problémov. Vďaka svojej komplexnosti vyžadujú modely dosahujúce najlepšie výsledky obrovské množstvo výpočtových zdrojov. Tieto požiadavky vedú k vývoju viacerých akceleračných techník zameraných na sprístupnenie týchto modelov na malých zariadeniach s obmedzeným množstvom zdrojov. Predmetom tejto diplomovej práce je akcelerácia neurónových sietí s cieľom redukcie počtu operácií násobenia reálnych čísiel.

Teoretická časť práce sleduje súčasné trendy a metódy využívané v oblasti akcelerácie neurónových sietí. Kvantizácia, odstraňovanie a zdieľanie parametrov zahŕňa techniky zamerané na redundanciu v presnosti a počte parametrov sietí. Binarizácia je extrémnou formou kvantizácie parametrov, pri ktorej sú váhy a prípadne aj aktivácie kvantizované na 1 bit. Takáto reprezentácia umožňuje SIMD spracovanie pomocou lacných logických alebo iných bitových operátorov. XNOR siete binarizujú váhy aj aktivácie podľa znamienka na hodnoty ± 1 a aproximujú presnosť pomocou skalárnych škálovacích parametrov. Táto transformácia umožňuje 32-násobnú redukciu veľkosti natrénovaných sietí pracujúcich s dátovým typom float a teoreticky až 58-násobné zrýchlenie konvolučných vrstiev. Binarizáčné techniky si napriek radikálnej strate presnosti na úrovni parametrov udržujú dobrú úspešnosť výstupu, čo podporuje domnienku o vysokej robustnosti súčasných architektúr. Najväčším problémom binarizovaných modelov je efektivita trénovacieho procesu. Stochastický gradientný zostup a všetky jeho rozšírené varianty vyžadujú vysokú presnosť parametrov. V súčasnosti sa tento problém rieši uchovávaním parametrov s vysokou presnosťou počas trénovania alebo viacbitovou kvantizáciou spätného prechodu.

Moderné nástroje umožňujúce návrh a implementáciu neurónových sietí, ako napríklad Torch alebo TensorFlow, nedisponujú funkcionalitou podporujúcou prácu s binárnymi tenzormi. Prvá časť praktickej časti rozširuje TensorFlow o implementácu kritických binárnych operátorov spustiteľných na GPU. Implementácia pozostáva z dvoch maticových binarizačných operátorov a z všeobecného maticového súčinu využívajúceho princípy XNOR sietí (XGEMM). XGEMM bol podrobený rýchlostnému benchmarku a porovnaný s cuBLAS verziou pracujúcou nad operandami s vysokou presnosťou. Napriek veľmi vysokej úrovni akcelerácie cuBLAS verzie, XGEMM dosiahol takmer 2,5-násobné zrýchlenie výpočtu. Implementovaný XGEMM je prvou dostupnou verziou XNOR všeobecného maticového súčinu, ktorá dosahuje citeľné zrýchlenie. Ďalšia optimalizácia môže viesť k výraznejšiemu zrýchleniu a potenciálne až ku komerčnej využiteľnosti.

Druhá polovica praktickej časti pozostáva z integrácie operátorov to jednoduchej doprednej siete. Zvolenou architektúrou bol 4-vrtstvový perceptron klasifikujúci ručne písané číslice (MNIST). Binarizované boli len 2 skryté vrstvy. Základnými faktormi ovplyvňujúcimi rýchlosť trénovania sú veľkosť trénovacej dávky a počet neurónov v skrytých vrstvách. V závislosti od týchto faktorov bolo dosiahnuté iba 25% maximálne zrýchlenie trénovania a priemerná strata úspešnosti o 5,5% na testovacej sade. Tento výsledok poukazuje na potrebu lepšej optimalizácie implementovaných operátorov. Benchmark nebol zameraný na udržanie vysokej úspešnosti klasifikácie. V poslednej kapitole sú porovnané state-of-the-art binarizačné metódy z pohľadu binarizácie. Binarizácia na menších datasetoch ako MNIST alebo SVHN dosahuje výborné výsledky. Nové metódy efektívneho návrhu a trénovania binarizovaných modelov výrazne zlepšili úspešnosť aj na náročnejších úlohách.

Neural Network Implementation without Multiplication

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Igor Szőke, Ph.D. All relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Bc. Lukáš Slouka
May 22, 2018

Acknowledgements

I would like to thank my supervisor Ing. Igor Szőke, Ph.D. for his help and support.

Contents

Chapter 1

Introduction

In recent years we have experienced a tremendous advancement in fields of visual and audio recognition tasks. Neural networks have been the tool that researchers had used to tackle these problems. However, existing models that achieve state-of-the-art results are extremely computationally expensive and in many cases, very memory dependent. In today's IoT world of small and embedded devices with limited memory and computational power, the resource requirements of such models are not feasible. Acceleration of neural networks is a go-to solution. The primary goal is to minimize network resource requirements without significantly hindering its performance.

Throughout the history of machine learning, researchers created many topologically distinct models that have different resource requirements. Probably the most well-known type suitable for media recognition tasks are convolutional neural networks (CNNs) which are a type of traditional feedforward networks. Acceleration of CNNs have been studied extensively during the past few years and researchers came up with several compelling acceleration techniques.

In this thesis, we delve deep into the problem of CNN acceleration. The text is subdivided into two logical parts. First part consisting of three chapters lays out theoretical foundation for network acceleration. The thesis starts with the detailed but straightforward description of feedforward neural networks and CNNs. The goal is to make sure that the reader understands where the computational and memory complexity originates. The second chapter provides insightful analysis of newest acceleration techniques, their applications, advantages, and drawbacks. Namely, we talk about parameter pruning and sharing, low-rank factorization, transferred/compact convolutional filters and knowledge distillation. The last chapter of the first part focuses in detail on binarization methods, which had the most success in limiting the number of expensive single precision multiplications in neural network computations.

The second part of the thesis focuses on practical implementations of binarized networks. A brief description of the TensorFlow framework is followed by a detailed specification of the implemented binary operators that have been missing in the framework. The most interesting is the GPU implementation of the general matrix multiplication using logical operators which achieves speed-up by a factor of 2.5 when compared to well optimized cuBLAS implementation. This result is extremely encouraging considering that the described implementation can still be improved significantly. The following chapter applies these operators in the image classifier and evaluates the performance. The last chapter focuses on binarized networks performance regarding the accuracy and provides several benchmarks of various network architectures for different tasks.

Chapter 2

Neural networks

Neural networks were first introduced in 1943 when McCulloch and Pitts published a model of neuron which worked as a simple threshold device to perform logic function [31]. Working principle of McCulloch-Pitts model stayed unchanged to this day. Neurons in networks can differ in their base[1] and activation[2] functions.

Networks of such neurons display remarkable ability to recognize patterns in the input data which in many cases outclass even human abilities. However, to accomplish such a feat, neural networks need to be sufficiently large and extensively trained resulting in memory and computational power requirements. To better understand these requirements, the next section describes a concept of feedforward artificial neural network (FNN) which is among the most commonly used architectural archetypes.

2.1 Feedforward networks

Neuron layers are the basic building blocks of the FNN's structure. Neurons within the same layer share base and activation function. Connections between the neurons are in a feedforward fashion, meaning there are no connections between neurons of the same layer and no feedback [12].

This text focuses on static networks with the constant number of neurons and their learnable parameters, that perform classification[3]. Following notations are used throughout the text to describe neural networks:

L the number of layers. Layers are indexed from 1

$\vec{\boldsymbol{a}}^{\boldsymbol{l}}$ output vector of layer l = input vector of layer $l+1$ ($\vec{a}_0$ is a vector of input features).

$\boldsymbol{W}^{\boldsymbol{l}}_{\boldsymbol{ij}}$ weight on the $j-th$ input of $i-th$ neuron in layer l. Size of weight matrix dimension $|\vec{W}^l_i| = |\vec{a}^{(l-1)}| + 1$ because of bias terms. Let $|W^l_{i0}|$ be the bias. Then by padding $\vec{a}$ so that $\vec{a}_0 = 1$ allows us to compute basis function as a single matrix multiplication.

$\boldsymbol{\theta(\cdot)}$ the activation function.

$\boldsymbol{J(\Theta)}$ the cost function.

[1] aggregating function of arbitrary number of neuron inputs
[2] threshold function with single input and output
[3] as opposed to regression problems, the task is to classify inputs into a predefined set of outputs

2.1.1 Fully-connected layer

Neurons in a fully-connected layer have connections to all activations in the previous layer. They share the number of the inputs which equals to the number of neurons in the previous layer or the neural network input width in case of the first layer. The process of computation of a fully-connected neural network is vectorizable into a single matrix multiplication operation. Equation 2.1 shows general case with a batch of activations and the weight matrix[4].

$$a^N = \theta\left(a^{(N-1)} \times W^N\right) \tag{2.1}$$

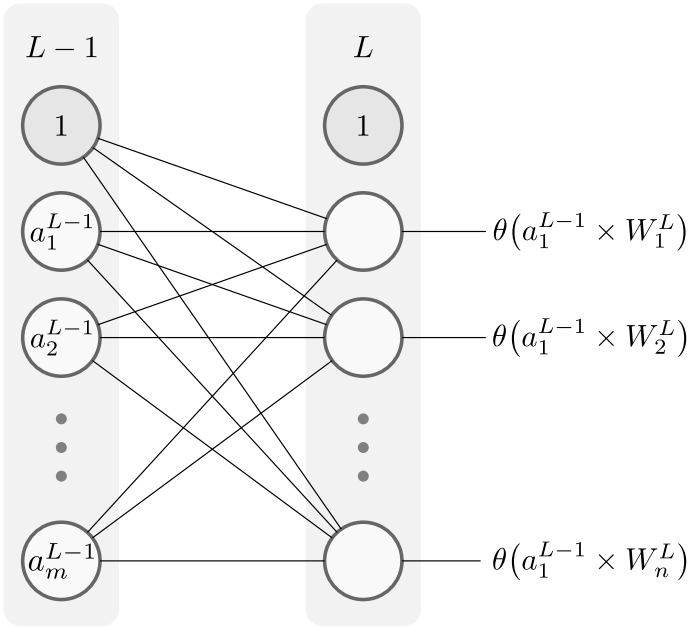


Figure 2.1: Visualisation of a fully-connected layer with n neurons and $m+1$ inputs.

The fully-connected layer consisting of n neurons with m inputs contains $n * (m+1)$ learnable parameters. This number could quickly reach millions or even billions when applied to image recognition. Fully-connected layers try to find patterns between all inputs. This behavior is useful only if there is minimal spacial relevance between individual inputs. Intuitively, this is not the case for audiovisual inputs, which have significant spacial dependencies and therefore using only fully-connected layers introduces many redundant parameters.

Sophisticated architectures use fully-connected layers to find non-linear combinations of high-level features, which are provided by more specialized layers such as convolutional or recurrent layers. Often the last layer of the model is a fully-connected layer.

2.1.2 Convolutional layer

Convolutional layers significanly minimize the number of learnable parameters by searching for patterns within selected spatial regions of the input. The spatial extent of this connec-

[4]the weight matrix is a column matrix meaning the i-th column holds weights of the i-th neuron

tivity is commonly known as the *receptive field* of a neuron or *filter size*. Connections of neurons are local along width and height dimensions of the input but always full along the depth of the input volume. For example, for an RGB input image and filter size equal to 5, there are $5 \times 5 \times 3 = 75$ connections for each neuron in the convolutional layer.

Output volume is defined by 3 hyperparameters:

Depth (N) Depth corresponds to the number of filters we want to use. Each filter looks for different patterns within local regions of the input volume. For example filters in the first convolutional layer may search for oriented edges or colored groups of pixels. Term *depth column* refers to a set of neurons that execute these filters in the same region.

Stride (S) Stride refers to the number of pixels by which we slide filters along the input volume. This parameter is rarely higher than 3.

Zero-padding (P) A technique used to control the spacial size of output volume. Zero padding of 1 adds a 1-pixel thick border of zeroes around the input volume.

For a 3-dimensional input volume $[W_1, H_1, D_1]$ a convolutional layer defined by N filters with the receptive field of size F, stride S and zero padding P produces output volume $[W_2, H_2, D_2]$ where:

$$W_2 = \frac{W_1 - F + 2 * P}{S} + 1 \tag{2.2}$$

$$H_2 = \frac{H_1 - F + 2 * P}{S} + 1 \tag{2.3}$$

$$D_2 = N \tag{2.4}$$

Convolution

Convolution operation is an iterative application of dot product of filter and selected input region. It might seem like an inherently iterative process without the option of vectorization. However, any convolution is transformable into a fully-connected layer with missing connections, in other words, a layer with non-learnable weights set to zero. This transformation requires an alteration of the input commonly known as *lowering*.

Consider input volume of size $[W, H, D]$ and convolution with N $[X, X, D]$-sized filters at stride S. This setup defines $L = LW * LH = (\frac{W-X}{S} + 1) * (\frac{H-X}{S} + 1)$ locations each of which has X^2D features. The process of convolution can be vectorized by following these steps:

1. Transform a location matrix into column vector with X^2D rows, and create a matrix LM composed of such column vectors for all locations ($|LM| = X^2D \times L$).
2. Transform filter matrices into row vectors of size X^2D and create filter matrix FM composed of such rows for all N filters ($|FM| = N \times Y^2$).
3. Calculate result of a convolution by computing matrix product $LM \times FM$.
4. Reshape resulting matrix of size $N \times L$ into desired volume of size $LW \times LH \times N$.

To have a better idea of a parameter scale in convolutional layers, see appendix C.

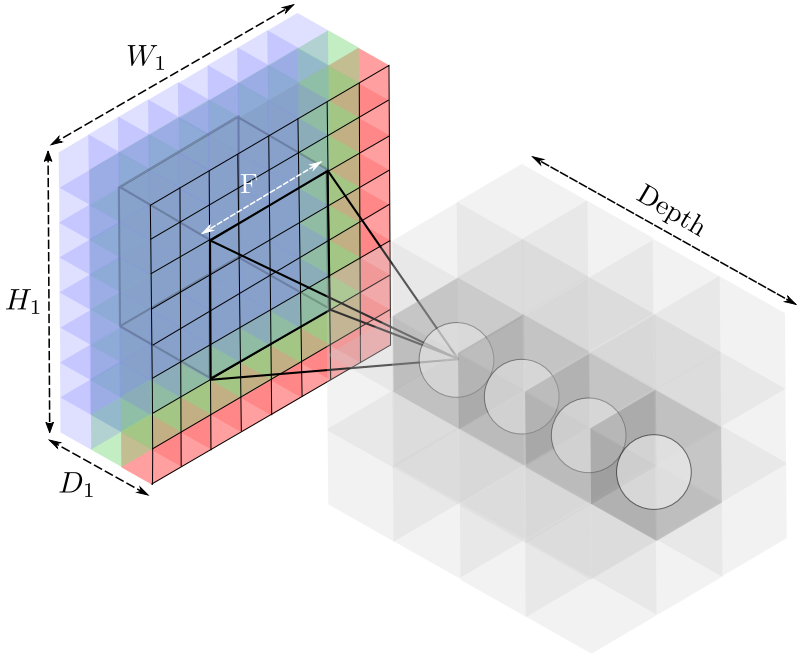


Figure 2.2: Convolutional layer with RGB input image of size $[8, 8, 3]$, filter size 4, no zero padding, stride 2 and depth 4. The image depicts depth column as the highlighted line of neurons in the convolutional layer. Total number of filters is $N = 3 \times 3 \times 4 = 36$.

2.1.3 Pooling layer

Pooling layers are used to reduce the size of output volume and are commonly inserted in-between convolutional layers. Pooling operation takes the local region in the input volume and selects one representative for each 2-dimensional slice of the input. Similar to convolutional layers, pooling layers are defined by their receptive field (F) and stride (S). A very common pooling has $F = 2$ and $S = 2$ with MAX operation, which reduces input volume by 75%.

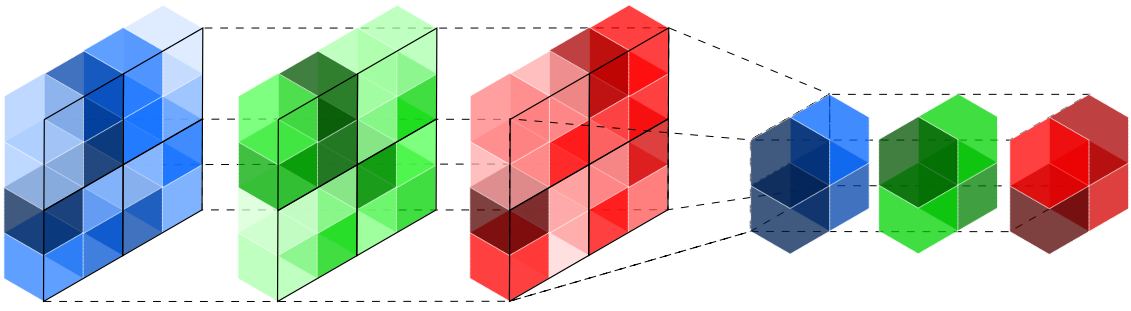


Figure 2.3: Visualization of 2 by 2 MAX pooling with a stride of 2 on $[8, 8, 3]$ input, that produces $[2, 2, 3]$ output.

Dilated convolutional layers[5] or convolutional layers with larger strides can be used to replace pooling layers [38]. This approach is essential in training of generative models, and it seems likely that the future architectures will feature less or no pooling layers.

2.2 Network complexity

In the previous section 2.1, we discussed feedforward networks and their common structural parts - layers. Most of the modern architectures stack these layers on top of each other to form complex deep neural networks. *Residual networks* (ResNets) achieved most recent state-of-the-art results. ResNets use dozens of convolutional layers organized into blocks of two followed by a single fully-connected layer. The two convolutional layers in a ResNet block are joined by a unit that adds input of the block with the output of the second convolution [18].

Size of the network model is often the determining factor concerning model's overall usability. Large models require more space and time to perform their function. Model space and time complexity are the primary measures used to evaluate acceleration techniques.

2.2.1 Space complexity

Memory is one of the most significant challenges in deep neural networks (DNNs) today. Limited memory bandwidth of DRAM devices is causing increased latency and power consumption when used for processing of vast amounts of weights and activations in neural network models. As shown in sections 2.1.1 and 2.1.2, the number of attributes can proliferate with each layer. For example ResNet 50 [18] with 50 convolutional layers requires over 95MB memory for storage for each image [4].

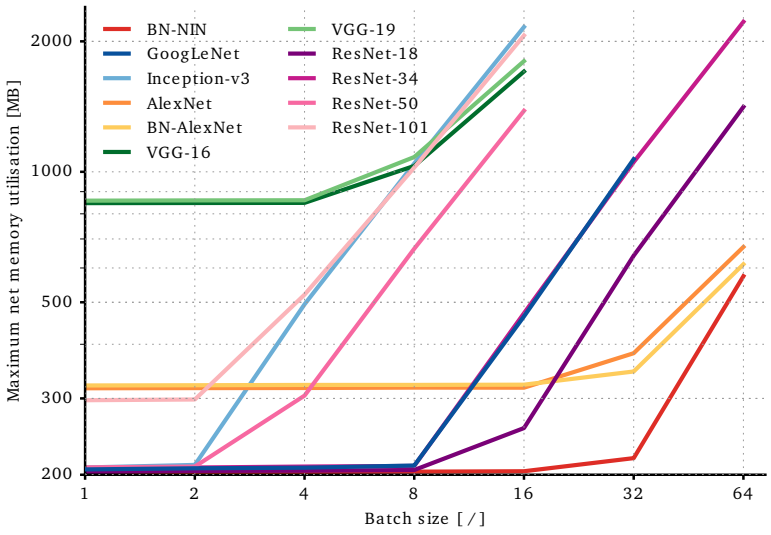


Figure 2.4: Memory requirements of various networks for the ImageNet challenge. All experiments were conducted on a JetPack-2.3 NVIDIA Jetson TX1 board with 256 GPU cores and 4 GB of shared RAM. Note that minimum size of 200 MB is caused by initial allocation of the network model. This measurements were taken by Canziani et al. [2]

[5] dilated layers perform convolution on dispersed regions. Closely connected regions have dilation of 0

GPU data requirements are challenging memory complexity as well. Vector paths in GPUs are typically 1024 bits wide which aligns with 32 samples in a batch of 32-bit floating point data. The result is an increase of a local storage requirement by a factor of 32. Lowering (see 2.1.2) is another example of an operation that boosts performance specifically on GPUs but increases memory complexity quadratically to the convolutional filter size. Measuring the memory use of ResNet 50 training with a mini-batch of 32 on a typical high performance GPU shows that it needs over 6 GB of local memory. Furthermore, high-performance graphics processors such as TitanX have only 1 KB of memory associated with each core that can be fast enough to saturate the floating point data path [10].

This lead researchers to come up with techniques to reduce the number of parameters so that the state-of-the-art networks can run on small embedded devices, cell phones, and FPGAs equipped with only several megabytes of memory[6].

2.2.2 Time complexity

In real life applications, there is a need for neural networks to produce results in real time. Good examples are cases of voice recognition and translation in real time during a conference call or visual recognition used by an industrial robot that sorts some product. These applications cannot afford to wait seconds to perform their tasks. Another consideration is the speed of training process. AlexNet in its original form took 2-3 days to train with GPU acceleration in place. Inference time is determined by the number of learnable parameters (determines the number of computations) and by the type of computation used. Common practices to optimize the type of computation include:

1. Use of less computationally expensive activation functions such as *(leaky) rectified linear units* (ReLU) over *sigmoid* or *tanh* activation functions.

2. Approximating single precision floating point operations with half precision, fixed point, low bit-width integer or even boolean operations with lower latency on both CPUs and GPUs.

3. Using a vectorized approach that takes advantage of fast matrix operations (e.g. lowering in 2.1.2).

Next chapter is dedicated to advanced acceleration techniques that substantially reduce both space and time complexity of deep FNNs.

[6] problem of compressing the reduced models is also relevant

Chapter 3

Advanced deep neural network acceleration methods

In this chapter, we take a look at new network compression and acceleration approaches as surveyed by Cheng et al. [4]. These approaches could be classified into 4 categories. *Parameter pruning, quantization, and sharing* aims to reduce performance-insensitive parameters in convolutional and fully-connected layers. *Low-rank factorization* focuses on the fact that filters in convolutional layers are typically 4-dimensional tensors with a significant amount of redundancy. These methods decompose filter tensors into tensors with lower rank and use them as an approximation of the original tensors. *Transferred/compact convolutional filters* based methods design special structural convolutional filters that reduce both space and time complexity. Lastly, *knowledge distillation* approaches learn a distilled model and train smaller neural network that approximates a more extensive network.

3.1 Parameter pruning, quantization, and sharing

Parameter pruning, quantization, and sharing approaches aim to reduce network parameters by removing redundant parameters or reducing them. We briefly touch this topic in Appendix C, where simple assumption[1] about the input data and their relevance towards convolutional filters lead to parameter sharing scheme that pruned millions of redundant parameters from the network. Based on the latest research, there are 3 classes of these techniques:

1. Model quantization and parameter binarization
2. Parameter pruning and sharing
3. Methods based on the structural matrix.

3.1.1 Quantization and binarization

Computations within neural networks are performed in a domain of real numbers, which translates into 2-byte half precision, 4-byte single precision or even 8-byte double precision arithmetic in computer implementations. Network quantization aims to compress the original network by reducing the number of bits per parameter, shrinking parameter domain

[1] filters in each depth slice/level can share weights because there is no precondition as to where edges or blobs of colors may appear on the image

during the process. The apparent result is a linear reduction of the space complexity, but possibly less noticeable is the time complexity optimization. Although GPUs are designed to perform fast floating point operations, quantization to lower bit-widths often leads to operations with smaller latencies. When it comes to quantization, it is a tradeoff between higher inference speed plus smaller memory consumption and overall network accuracy.

Quantization

In 2013, Denil et al. demonstrated that the weights within one layer could be accurately predicted by $\sim 5\%$ of the parameters, which indicates that the neural network is overly parameterized. That inspired the use of vector quantization methods, such as:

- Application of k-means scalar quantization to the parameter values [13].
- 8-bit parameter quantization [45].
- 16-bit fixed-point representation in stochastic rounding based convolutional neural network training [15].
- Minimizing Hessian-weighted quantization errors in average for clustering network parameters [5].
- Deep three-stage compression with codebook encoding [17] (state-of-the-art).

Quantization performs the division of the input space into several connected regions called *Voronoi regions/polygons* [12]. In more practical terms, quantization maps larger input space onto more compact and more easily representable output space. Quantization is often accompanied by clustering, since many activation functions, including rectified units, have a range of $\mathcal{R}$ values. Appendix D contains a description of the state-of-the-art deep three-stage compression method which uses quantization with k-means clustering to reduce network size by a factor of ≈ 30.

Binarization

Binarization methods use weights quantized and projected down to 2 values. Modern approaches tend to use binary representations even during the training. As [32] showed, such networks are remarkably robust. These networks provide maximum weight compression as well as ability to use binary operations instead of expensive floating point operations. Furthermore, such networks are an excellent choice for hardware implementations. Chapter 4 describes theory and math behind binarization techniques in detail. Practical implementations of binarized models are subject of chapters 5 to 7.

3.1.2 Parameter pruning and sharing

Network pruning and sharing have been used to reduce network complexity and to address the over-fitting issue. These methods minimize the number of connections by removing ones with low impact or by sharing same weights in between neurons. Early approaches removed excess weights based on the Hessian of the cost function (e.g. OBN[2] [9]).

Work by Srinivas and Babu [39] proposed pruning those weights which change the output neuron activations the least. An approach known as *HashNets* uses low-cost hash functions to create weight hash buckets for parameter sharing [3].

[2] optimal brain damage

3.1.3 Structural matrix

Previously mentioned methods focused primarily on convolutional layers. However, there is an abundance of redundant parameters within fully-connected layers, which are still a substantial part of the most modern architectures such as residual networks. Billions of parameters and computations within fully-connected layers are the most memory consuming parts of networks that feature them. The bulk of fully-connected layer computations are in general performed by weight matrix W of size $m \times n$ and input vector $\vec{a}$ multiplication.

Structural matrix based method try to find structure in W, such that it can be described with less than mn parameters. One of the most recent methods proposes use of Adaptive Fastfood Transform [48], which with clever use of random diagonal matrices, random permutation matrix, Walsh-Hadamard matrix and fast Fourier transform manages to reparametrize W so that memory complexity changes $\mathcal{O}(nm) \to \mathcal{O}(n)$ and computational complexity drops $\mathcal{O}(nm) \to \mathcal{O}(n \log m)$.

3.2 Low-rank factorization

Low-rank factorization methods accelerate (convolutional) neural networks via minimizing the rank of matrices representing convolutional filters. Recall that input and output of a typical convolution layer are 3D tensors 2.1.2. Convolution kernel itself is a 4D tensor. The primary solution strategy for rank minimization has been based on nuclear-norm minimization which requires computing singular value decompositions which gets more costly with larger matrices and ranks [47]. The following equation defines low-rank decomposition of matrix A of size $n \times m$ with rank R:

$$A(i,j) = \sum_{r=1}^{R} A_1(i,r)A_2(j,r), \qquad i = \overline{1,n} \quad j = \overline{1,m} \tag{3.1}$$

Two approaches have stood out in this domain of acceleration in recent years.

1. Canonical Polyadic (CP) decomposition [26]
2. Exact closed form with batch normalization (BN) decomposition [43]

3.2.1 CP decomposition

Canonical polyadic decomposition is the most straightforward way to separate variables in case of multiple dimensions. For a d-dimensional matrix A it builds on top of equation 3.1 and has the following form:

$$A(i_1, \ldots, i_d) = \sum_{r=1}^{R} A_1(i_1, r) \ldots A_d(i_d, r) \tag{3.2}$$

Lebedev et al. [26] use CP-decomposition in tandem with non-linear least squares to replace each convolutional layer by a sequence of 4 convolutional layers with much smaller kernels. Resulting network is optimized via the regular backpropagation process. This approach managed to achieve speedup by a factor of 4 on a second convolution layer of AlexNet (C).

3.2.2 BN decomposition

This approach builds directly on top of 2-way tensor decomposition proposed by Jaderberg et al. [22] and improves it by proposing a new algorithm for computing low-rank decomposition and a new method for training low-rank constrained convolutional neural networks.

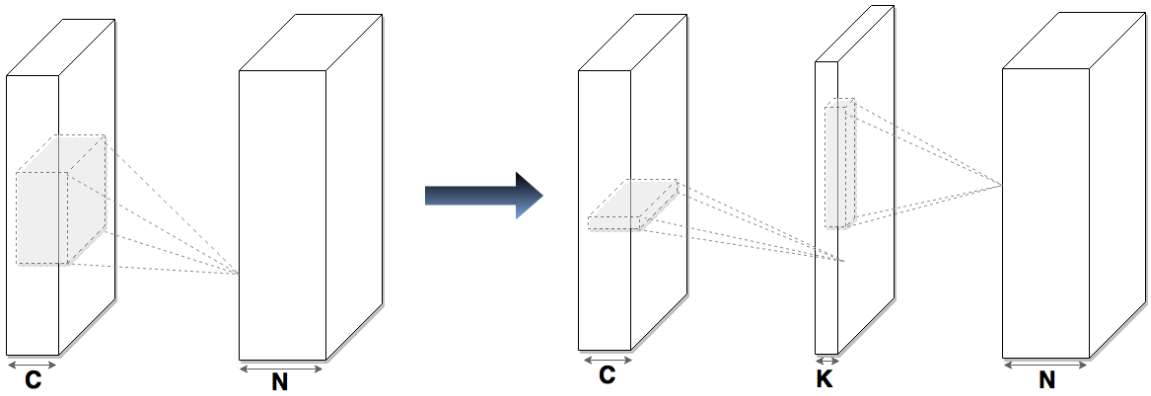


Figure 3.1: 2-way decomposition. The original convolutional layer is on the left. Low-rank constraint convolutional layer with rank-K is on the right.

As mentioned in 3.2, computing low-rank decomposition is difficult, and in general, it is done by iterative schemes. BN decomposition finds the exact closed-form solution for a particular form of low-rank decomposition. This computation is much more effective than previously used iterative methods. During the training process, convolutional kernels are parametrized in a way that enforces the low-rank constraint. This is complemented with *batch normalization* training technique proposed in 2015 by Ioffe & Szegedy [21].

3.3 Transferred/compact convolutional filters

Research by Cohen et al. in [6] has motivated recent advances in transferred convolutional filters. The research stated the concept of equivariance based on a significant amount of empirical evidence. Equivariance states that for an input tensor x, network or a layer $\Phi(\cdot)$ and a transform tensor $\mathcal{T}(\cdot)$ the following formula holds:

$$\Phi(\mathcal{T}_g x) = \mathcal{T}'_g \Phi(x) \tag{3.3}$$

That is, transforming an input x by a transformation g and then passing it through the learned map Φ[3] should give the same result as first mapping x passed through Φ before the final transformation of the representation. The takeaway idea is that it is reasonable to perform transformations on layers or even convolutional filters.

Building on that idea, several works have been published that proposed convolutional layers defined by a set of base filters and transformations that apply to them. For example, the work in [11] exploited the cyclic symmetry of filters in convolutional layers by generating new filters via 90° rotations and horizontal/vertical flipping of base filters.

[3] learned network or layer

3.4 Knowledge distillation

The core idea of this approach is modeling and training more dense networks to mimic deep neural networks. The work in [1] and [19] showed that shallow neural networks of modest size (student) could approximate the function performed by larger networks (teacher) trained on the input data set. The teacher network produces scores on the input, which are in turn used to train the student network. As of now, it is not possible to train student model directly on the original input so that the model reaches the same accuracy as either learned teacher nor learned student network.

Romero et al. [35] further extended the student-teacher paradigm in a work which proposed training deep but thin networks called *FitNets*, that made the student mimic the full feature maps of the teacher.

In general, knowledge distillation approaches achieve significant computation time reduction. However, they are applicable only to classification tasks with the softmax loss function.

Chapter 4

Binarized networks

Previous chapters discussed the concept of neural networks concerning memory and computational complexity (chapter 2) and surveyed advanced acceleration strategies that tackle the problem of redundancy and computational overhead within the network architectures (chapter 3). In section 3.1.1 we introduced network quantization techniques and the extreme case of network binarization which is the most suitable strategy for reducing the number of multiplication operations in the network.

Network binarization faces difficult challenges on both theoretical and practical level. Keeping high precision of weights during the parameter update is paramount for the stochastic gradient descent to work correctly. Parameters in pure binarized networks have no precision at all. As a result, current solutions tend to keep floating point approximations of parameters or their quantized equivalents during the training and re-binarize after each parameter update.

Several approaches in the past tried to simplify computations. Using powers of 2 and reducing multiplications into binary shifts were among the first. These methods practically eliminated multiplication, but experienced a significant drop in model accuracy and failed to guarantee training convergence. Binarization reduces the range of weight values down to 2 options. Typical cases would be $\{-1, +1\}$ or $\{0, 1\}$. In 2015 Courbariaux et al. [8] proposed training method called *BinaryConnect* which substitutes binary weights during both forward and backward propagation in feedforward networks. This approach was later improved on by Lin et al. [30], which introduced *TernaryConnect*.

Even though BinaryConnect managed to achieve excellent results on the smaller scale (MNIST, CIFAR-10, SVHN), it lacks on large-scale datasets. Advances achieved by Rastegari et al. [34] outperform BinaryConnect based methods and are presented later in this chapter in section 4.2.

From the practical standpoint, modern machine learning frameworks such as TensorFlow or Torch have not implemented support for binary tensors yet. Available implementations of binarized networks are only proofs of concepts.

4.1 BinaryConnect

As shown in the first chapter, activation a_i^l of an i-th neuron of l-th layer is the output of the neuron activation function with input in the form of the scalar vector product of input vector $\vec{a}^{l-1}$ and weight vector $\vec{W}_i^l$.

$$a_i^l = \theta(\vec{a}^{l-1} * \vec{W}_i^l) = \theta\Big(\sum_{j=0} a_j^{l-1} * W_{ij}^l\Big) \tag{4.1}$$

BinaryConnect (BC) constraints the weights to be either $+1$ or -1 during propagations. This means that equation 4.1 can be expressed as a difference of 2 sums:

$$a_i^l = \theta\Big(\sum a^+ - \sum a^-\Big) \tag{4.2}$$

a^+ and a^- are sets of previous layer activations with positive/negative connection weights. By using the rectified linear unit (ReLU) as the activation function which has a form of $\theta(x) = max(0, x)$, this approach completely removes multiplication in the forward pass.

4.1.1 Weight binarization

Binarization of weights is the operation that transforms weights of arbitrary type in the original network into binary values. BC suggest two binarization techniques.

Deterministic binarization transforms weights by their sign. Alternatively, we could introduce a threshold which would be the average of an input vector, which could compensate for the loss of information.

$$w_b = \begin{cases} +1 & \text{if } w \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.3}$$

Stochastic binarization uses hard sigmoid function σ to calculate the probability of the weight binarization as either one of the two values.

$$\sigma(x) = \begin{cases} +1 & \text{if } x \geq 1 \\ 0 & \text{if } x \leq 0 \\ \frac{x+1}{2} & \text{otherwise} \end{cases} \tag{4.4}$$

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{4.5}$$

This approach is more delicate and more correct averaging process, which yields better results in experiments [8]. However, stochastic binarization brings the problem of efficient generation of random numbers. That gets partially solved by the mini-batch nature of the learning process where binarization of inputs is performed only once for the whole mini-batch compared to the individual multiplication for each entry in the mini-batch.

4.1.2 Quantized backpropagation

A proposed approach to removing multiplications from the backward pass in [30] uses the power-of-2-quantization which allows replacing multiplications by binary shifts. During the backward pass, error signal δ propagates from the output layer back towards the first

input layer. Along the way, for each layer with weight tensor W, bias tensor b, precomputed gradient tensor of the layer output ∇a and layer input x, updated weight tensor is calculated as:

$$\Delta W = \eta\big(\delta \odot \nabla a\big) * x^T \tag{4.6}$$
$$\Delta b = \eta\big(\delta \odot \nabla a\big) \tag{4.7}$$

η is the learning rate and operator $\odot$ is element-wise multiplication. Error signal δ is updated via the following rule:

$$\delta = \big(W^T \delta\big) \odot \nabla a \tag{4.8}$$

Since input tensor x (output tensor a) has a known distribution of values, it is plausible to perform quantization of each entry to an integer power of 2. That way outer multiplication in equation 4.6 which represents the bulk of multiplications, can be replaced by bit shifts. Experiments in [30] showed, that 3 bits are enough to quantize x. Amount of the remaining multiplication operations is negligible.

4.1.3 Parameter update

Out of 3 training phases, weights are binarized only during the forward and backward propagation. Keeping good precision weights during the parameter update is paramount for the stochastic gradient descent to work at all. This requirement implies a need to keep high precision weights in memory. Therefore memory complexity is not optimized.

BC forces weights to be real values in the interval $\langle -1, 1\rangle$ ensuring that binary probabilities are in a reasonable range of values. If during the parameter update a weight value escapes this range, it is clipped at the edges of the interval.

4.2 XNOR-Networks

In 2016 Rastegari et al. [34] published an alternative approach towards binary networks. They proposed 2 solutions. Binary weights are characteristic of a simpler *Binary-Weight-Network* (BWN) model with real value inputs. More advanced *XNOR-Network* (XNN) in addition to binary weights transforms real value input into binary input and use XNOR bit-counting operation to compute convolutions.

4.2.1 Binary weight networks

As mentioned previously, the goal of quantization is a precise approximation. Consider weight filter W in a CNN. BWN approximates W with binary weight filter B and scaling factor $\alpha \in \mathcal{R}^+$ such that $W \approx \alpha B$. Approximation precision is maximized[1] by minimizing the squared norm of the difference of W and αB.

$$\alpha^*, B^* = \underset{\alpha, B}{\operatorname{argmin}} \left(\|W - \alpha B\|^2 \right) \tag{4.9}$$

By solving the equation 4.9, the optimal estimation of weight filter is achieved by taking the sign of weight values $\implies B^* = \operatorname{sign}(W)$, and the optimal scaling factor is the average

[1] α^*, and B^* are optimal approximations

of absolute weight values $\implies \alpha^* = \frac{1}{n}\|W\|$ where n is the number of weights in the filter W. Then for convolution $*$ with input tensor A the following formula holds:

$$A * W \approx (A \oplus B)\alpha \tag{4.10}$$

Operation $\oplus$ is convolution with additions and subtractions only, similar as in 4.1. Note that multiplication is not entirely removed in the forward pass, as it still needs to be scaled by α.

Training algorithm

Algorithm for BWN training follows similar steps as for BC.

1. For each filter in every layer compute scaling factor α and binary filter B. Approximated weight filter is then $\widetilde{W}$.
2. Perform standard forward propagation using equation 4.10 which results in a prediction tensor $\widehat{Y}$ and cost J.
3. Compute gradients as $\frac{\partial J}{\partial \widetilde{W}}$.
4. Use standard SGD or ADAM to update the original weight tensors using computed gradients of their approximations and learning rate η.
5. Update learning rate via any learning rate scheduling function.

Analysis

After the training completes, real value weight tensors are no longer required because forward propagation requires only binary weights and scaling factor. This approach achieves acceleration only during the forward pass. Training achieves no memory saving. However, a trained network operates with $\sim$ 32x memory saving and $\sim$ 2x time-saving. BWN proved to be successful even on more extensive networks such as AlexNet, where it performed without any loss of accuracy.

4.2.2 Input binarization

Unsurprisingly, the process of input binarization follows the same idea as weight binarization. However, the main focus is not necessarily on precise approximation of the tensor, but on precise approximation of entire convolution operation such that for any input tensor A and weight filter W we find optimal binary tensors B_A and B_W and scaling factors α and β such that:

$$A^T * W \approx \beta(B_A)^T * \alpha B_W \tag{4.11}$$

That leads to a similar optimization problem as in BWNs, with the solution of $B_W^* = \text{sign}(W)$, $B_A^* = \text{sign}(A)$ and scaling factors α and β being the averages over absolute values in tensors W and A respectively. As mentioned in [7], convolving 2 binary tensors can be performed by XNOR and bit-counting operations. Consider input and weight tensors of 8 binary inputs. Then by concatenating these tensors into two 8-bit integers, we can perform dot product on these values with only 2 inexpensive instructions (figure 4.1). Equation 4.12

shows the relationship between vector dot product and XNOR dot product, where n is the size of the input vectors X and W.

$$A \cdot W = 2 * \text{popcount}\big(A \text{ xnor } W\big) - n \tag{4.12}$$

Doubling of the population count result can be implemented as a left shift by one position, which completely removes any form of multiplication from the calculation.

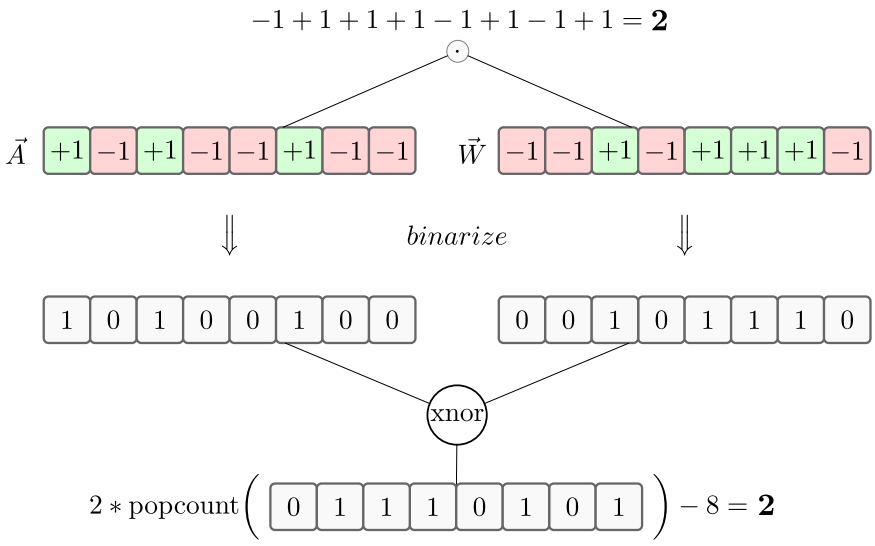


Figure 4.1: Dot product of two binary vectors $\vec{A}$ and $\vec{W}$ performed by XNOR and popcount single-instruction operations. Popcount returns the number of non-zero bits.

4.2.3 Binarized convolution

If we consider standard use-case of the convolution operation, we end up with a relatively small 3D kernel (width × height × depth) on a relatively large 3D input of the same depth. Input scaling factor is computed for all input sub-tensors that align with convolution kernels and steps. Overlaps of the sub-tensors during convolution steps require many redundant computations. The algorithm is broken into 2 parts to avoid this redundancy. In the first phase, we flatten the input by averaging its elements along the depth axis. Flattened matrix (A_F) when convolved with constant matrix k results in a matrix K that contains all the necessary scaling factors β (figure 4.2). Equation 4.13 shows relationship between full precision convolution and its binary approximation ($\cdot$ marks element-wise product).

$$A \odot W \approx \Big(\text{sign}(A) \oplus \text{sign}(W)\Big) \cdot K\alpha \tag{4.13}$$

4.2.4 XNN training

Typically, convolutional networks use several different layers to complement the convolutional layer. The output of the convolution is normalized with batch normalization [21], activated via a non-linear function, and pooled afterward. This sequence of transformations does not work well with binary networks. Applying normalization on binary tensors

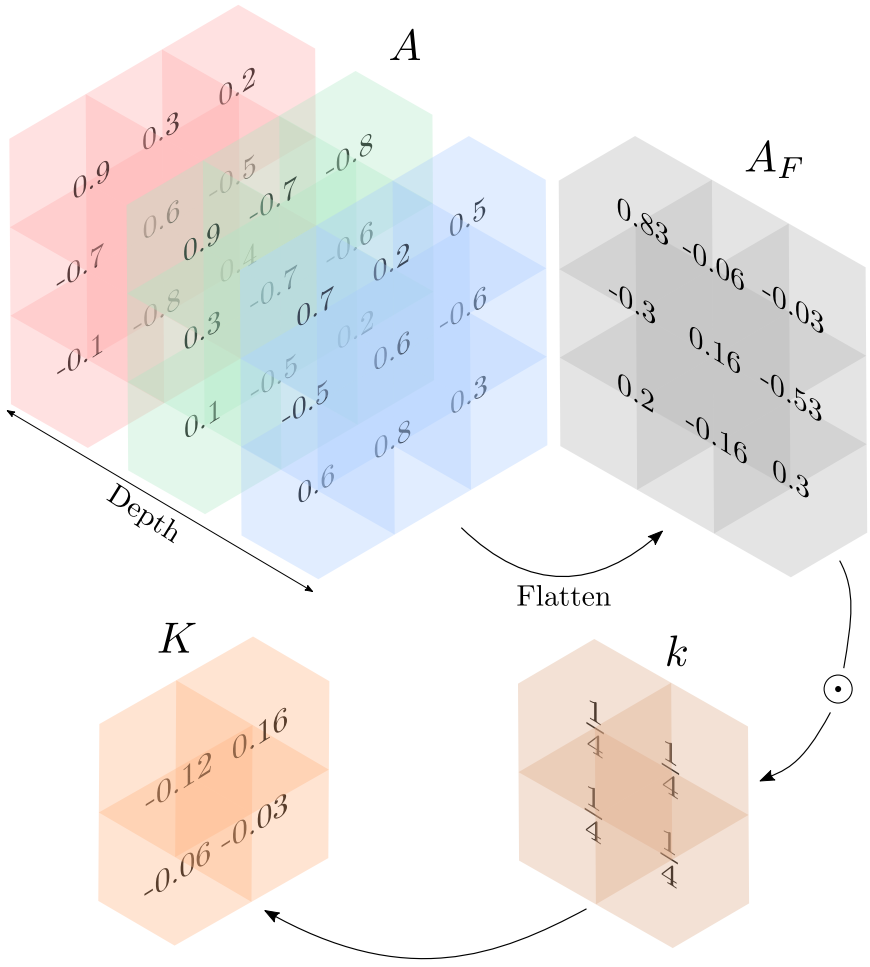


Figure 4.2: Optimized β-matrix computation.

does not affect them, and pooling causes significant loss of information. Rastegari et al. normalized and activated the input tensor before binary convolution. They proposed using non-binary activation after the binary convolution on state-of-the-art networks. The last step in the sequence could be any pooling.

Chapter 5

XNOR kernel implementation

So far we have discussed the theory behind neural network binarization. This chapter focuses on the implementation of binarization principles in TensorFlow (5.1) framework which has an excellent extensible developer interface that allows creating new operators from scratch and integrating them with the rest of the framework in a seamless fashion. To port XNN principles into the framework, we had to create operations for binarization of tensors and a general matrix multiply (GEMM) operator that utilizes only the xor, not and population count instructions to perform the matrix multiplication. These operators have to compete with cuBLAS and Eigen3 GEMM implementations used in TensorFlow which are exceptionally well optimized and beat them to achieve a performance speedup.

Next section briefly describes the TensorFlow framework and what it takes to extend it by new operators. Later sections in this chapter describe in detail the implementation of binarization and XNOR GEMM (XGEMM) and compare them to the TensorFlow matmul operator.

5.1 TensorFlow

Tensorflow is an open source library for high-performance numerical computation generally used in machine learning applications. The core principle is building an interconnected graph of operators that perform various mathematical transformations on input tensors. Tensors are n-dimensional vectors of typed numbers. Supported types range from 8-bit real numbers up to the 64-bit complex numbers with signed and unsigned variants. TensorFlow comes with TensorBoard, which is an application for visualization and statistical evaluation of TensorFlow graph executions.

Neural networks generally contain dozens if not hundreds of mathematical operations that are hard to visualize and difficult to debug. Tensorflow allows to group up related operators by arbitrary relation, such as involvement in the same layer. This grouping makes graph visualization in TensorBoard clearer, and it also provides more comprehensive statistics.

5.1.1 Extending the framework

Each operator consists of C++ and CUDA backend and operation's frontend. TensorFlow declares operation interface (inputs, outputs, data types, additional attributes and shape inference) via the C++ `REGISTER_OP` call. Operation name links the frontend declaration with the operation's backend.

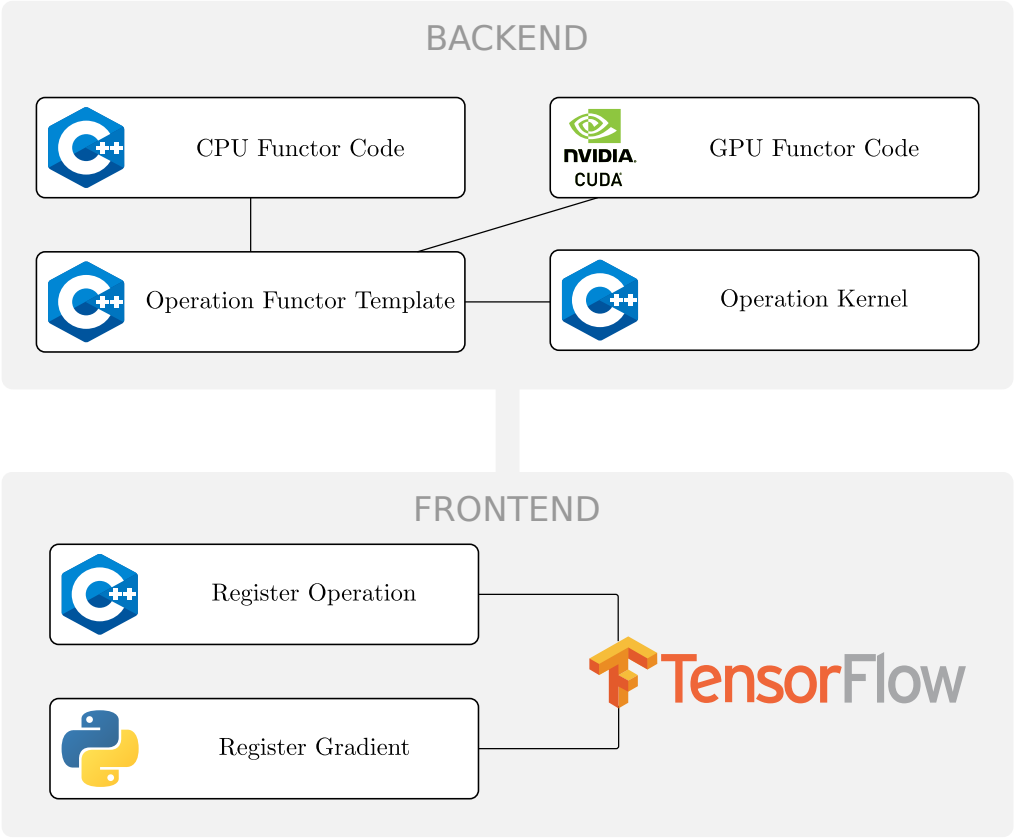


Figure 5.1: Steps required to create and integrate a new TensorFlow operator.

Operation kernel is the core of operation's backend implementation. Kernel checks the validity of inputs and prepares data structures for the outputs in the memory. It also calls functors, the templated execution units of operations. CPU functors usually embed CPU implementation in their body. GPU functors prepare CUDA thread distribution and execute CUDA kernel. Operations used in the neural networks usually need complementary gradient computation. TensorFlow allows implementing gradient calculation via the `RegisterGradient` decorator directly in python.

Diagram in figure 5.1 sums up and visualizes the process of extending the TensorFlow API with a new operator. It is important to note that TensorFlow does not insist on this structuring and it should be used only as a template of best practice[1].

5.2 Tensor binarization

Unfortunately, TensorFlow does not support 1-bit tensors, and the matrix multiplication operator requires half, single and double precision floating point numbers, complex numbers with minimal bit-width of 32 bits or 32-bit integers as operands. The consequence of this is that implementing XGEMM as a combination of low bit-width operands with standard GEMM to achieve a speedup is not an option.

[1] visit TensorFlow tutorial pages for more information

TensorFlow provides a suite of bitwise operators for tensors with integral types. Therefore the best course of action was to pack batches of N elements into a single number with integral type with bit-width of N bits, which would allow processing of N elements at once with bitwise operators. My implementation focuses on binarizing matrices since they are the inputs of matrix multiplication methods and cover most of the need in neural network models.

Next two sections focus on GPU implementation of two matrix binarization approaches. The row-wise approach which reduces matrix width and column-wise transposition binarization which combines matrix transposition and row wise binarization into one operation.

5.2.1 Matrix row binarization

The matrix row-wise binarization operator takes an input matrix and binarizes it row by row. Allowed input data types are 8, 16 or 32-bit integers and 16, 32 or 64-bit floating point numbers. The output is determined by the `qtype` attribute which has to refer to an integral data type.

Row-wise binarization operation kernel checks the input and allocates memory for the output. The number of output rows stays the same and output columns are calculated according to the equation 5.1, where C_I is the number of input columns and B is bit-width of `qtype`.

$$C_O = floor\left(\frac{C_I + B - 1}{B}\right) \tag{5.1}$$

GPU functor organizes threads into blocks of 64 arranged in an 8×8 matrix. Each thread computes one element of the output processing B elements of the input. Figure 5.2 visualizes the process.

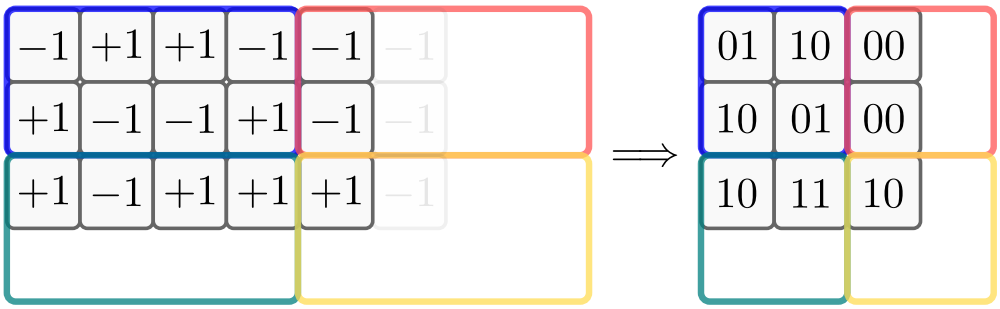


Figure 5.2: Row binarization with $B = 2$, block size $= 2$ and an input matrix with size 3×5.

5.2.2 Matrix column binarization

To achieve maximum efficiency and the best speed up, every operator had to be implemented on a rather low level. The most time-consuming part of an operation executed on big chunks of data is memory access [36]. Data access in a row-wise fashion as in 5.2.1 is fast thanks to the prefetcher which loads adjacent memory locations in advance. When accessing elements in a column-wise manner, we jump across the memory by a whole row each time losing the benefit of prefetched data in the process.

This issue comes in a column binarization as well as in matrix multiplication. Transposing the second operand allows matrix multiplication algorithm to access both matrices in a row-wise fashion. Column binarization operator combines matrix transposition with binarization making the data preparation process of GEMM faster.

Current implementation takes the same input arguments as 5.2.1, but the output is restricted to 32-bit integers. Operation kernel prepares output matrix dimensions as in equations 5.2, where R references rows, C columns and subscripts I and O refer to input/output matrix.

$$\begin{aligned} R_O &= C_I \\ C_O &= \mathit{floor}\left(\frac{R_I + 31}{32}\right) \end{aligned} \tag{5.2}$$

CUDA kernel operates in 3 phases. During the initial phase, each thread in a block sets one row of the shared memory (32 cells) to a negative one value. This step ensures that during the binarization, the exceeding values are zeroed out. With shared memory correctly initialized, the warp of threads proceeds to fill shared memory with the input matrix values. Each thread tries to load 32 values into a single column of the shared memory. Notice that the shared memory contains one transposed block of the input matrix. The third phase performs binarization of each column of the shared memory and saves the result to a cell of the output that corresponds to the position of the thread in CUDA thread matrix. The whole process is visualized in figure 5.3.

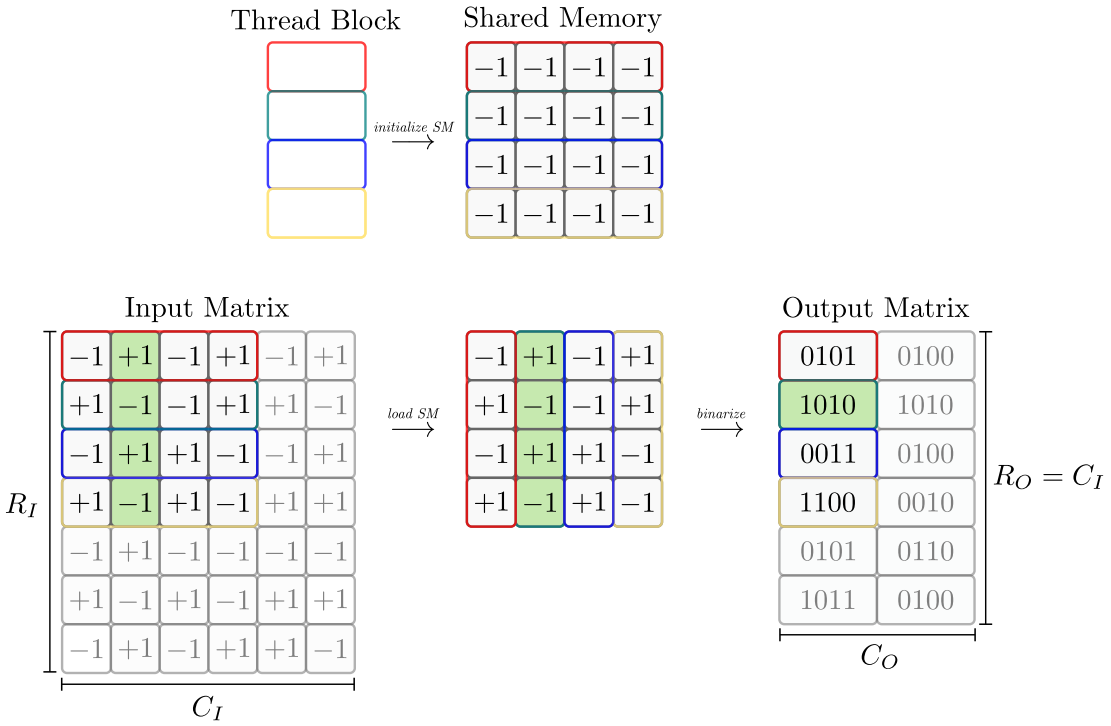


Figure 5.3: Column binarization with transposition of an input matrix with size $R_I \times C_I$. Binarization bit-width is same as thread count per block = 4.

5.3 XGEMM

XNOR general matrix multiply operator follows the same principles as traditional matrix multiplication operator. Matrix multiplication $C = A \times B$ is a $\mathcal{O}(n^3)$ operation that composes the output matrix element c_{ij} as a dot product of the i-th row of A and j-th column of matrix B (equation 5.3).

$$A_{i,j} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,j} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,j} \end{bmatrix} \qquad B_{j,k} = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{j,1} & b_{j,2} & \cdots & b_{j,k} \end{bmatrix}$$

$$C_{i,k} = A \times B = \begin{bmatrix} \sum_{x=1}^{j} a_{1,x}b_{x,1} & \sum_{x=1}^{j} a_{1,x}b_{x,2} & \cdots & \sum_{x=1}^{j} a_{1,x}b_{x,k} \\ \sum_{x=1}^{j} a_{2,x}b_{x,1} & \sum_{x=1}^{j} a_{2,x}b_{x,2} & \cdots & \sum_{x=1}^{j} a_{2,x}b_{x,k} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{x=1}^{j} a_{i,x}b_{x,1} & \sum_{x=1}^{j} a_{i,x}b_{x,2} & \cdots & \sum_{x=1}^{j} a_{i,x}b_{x,k} \end{bmatrix} \tag{5.3}$$

The naive implementation of matrix multiplication (3 nested loops) suffers from discontinuous memory access and thus from an ineffective utilization of cache memory units. The following subsection describes several matrix multiplication techniques most of which are used to optimize operations on multidimensional tensors. Matrix preprocessing, usage of shared memory and tiling are among the ones that optimize the implemented CUDA kernel.

Several additional challenges are raised as a result of the usage of XNOR based computations. One of the problems is the impact of necessary data padding, which is caused by the fact that any combination of allowed weights in XNNs affects the output. Subsection 5.3.2 defines the problem more closely and describes implementation steps taken to solve this issue.

5.3.1 GEMM optimization

Matrix multiplication is a notorious optimization problem. GPUs are well suited to compute this operation fast thanks to the density of the operation's inputs. There is a myriad of optimization techniques that improve matrix multiplication performance. This section focuses on the most powerful methods such as tiling, global memory coalescing, prefetching and loop unrolling.

Tiling

One of the most significant problems of algorithms operating on large data structures is that after a certain boundary the cache loses its ability to prefetch data from the main memory efficiently and to maintain temporal data locality. That is a cause of the performance drop on more massive inputs. Tiling, also known as blocking is a go-to solution for this problem. One of the advantages of this approach is inherent compatibility with matrix processing on GPUs.

Tiling increases a computation-to-memory ratio by dividing the output matrix into rectangular parts called tiles. Usually, the tiles are 16×16 or 32×32 squares. Tiled algorithm loads tiles from both input matrices into separate buffers and computes the change of the output for each element of the tile. Tiles from matrix A are loaded in a row-wise pattern and tiles from B in a column-wise pattern. Tile row index of A defines row index of C and tile column index of B defines column index for C. Figure 5.4 visualizes the process.

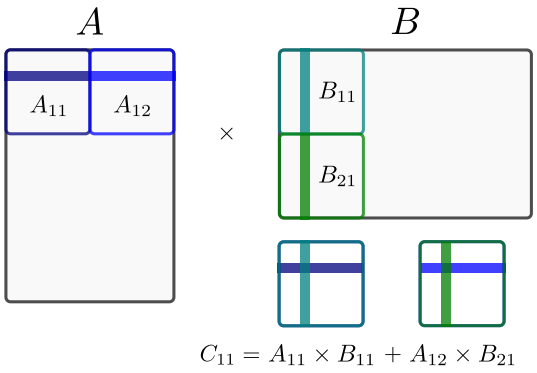


Figure 5.4: Tiled matrix multiplication.

Tiling is an easily parallelizable algorithm on both GPUs and CPUs. Each thread in a block (tile) computes one element of the output matrix.

Global memory coalescing and avoiding shared memory bank conflicts

These optimizations are related to GPU implementations. All threads in a warp execute the same instruction. If that instruction loads data from the global memory, hardware checks whether threads access consecutive memory locations. If that is the case, hardware coalesces all memory accesses into consolidated access to consecutive DRAM addresses.

Another thing to keep in minds is that shared memory on the GPU has 32 banks. Successive addresses are assigned to successive banks. Shared memory bandwidth is 4 bytes per bank per clock cycle. Designing an algorithm to avoid bank conflicts helps to achieve higher memory throughput while decreasing computation time.

Prefetching and loop unrolling

Prefetching speeds up computation by explicitly requiring data in advance so by the time they are needed they are already prepared in the cache. Hardware has prefetching units that can recognize simpler patterns in the access. However, software prefetching is used with tiling to load data for the next tile computation into registers while the current tile is computed.

Loop unrolling is a counter-intuitive process of increasing loop step and multiplying code in the loop body. Unrolling can reduce the number of required instructions (especially branching) and reduce the execution time. Intelligent compilers automatically unroll inner loops. Outer loops can be unrolled manually or with `#pragma unroll` directive.

5.3.2 Implementation

Current XGEMM operator accepts input matrices of any integral type and produces 32-bit integer, single and double precision floating point output. Operation kernel checks the validity of both input matrices and prepares them for XGEMM functor.

Input preprocessing

XGEMM operation kernel uses internal binarization on a 32-bit width. The kernel uses row-wise binarization functor to binarize first input matrix A and column-wise transpose functor to binarize the second input B. The number of columns in the A matrix must be the same as the number of rows of the B matrix. That number represents the size of binarization axis for both binarization functors. Final dimensions of binarized matrices A_{bin}, B_{bin}, and the final output matrix C are:

$$\begin{aligned} Col_{A_{bin}} = Col_{B_{bin}} &= \frac{Col_A + 31}{32} = \frac{Row_B + 31}{32} \\ Row_{A_{bin}} &= Row_A = Row_C \\ Row_{B_{bin}} &= Col_B = Col_C \end{aligned} \tag{5.4}$$

Transposition of the second matrix partially solves the problem of global memory coalescing in the XGEMM CUDA kernel.

Data padding

Tiled computation of matrix multiplication results in internal padding whenever matrix dimensions are not integer multiplications of the tile dimensions along both axes. For a standard floating point GEMM, this does not cause any issues. Tiles are zero padded, which results in several zero additions that do not affect the output.

XGEMM does not use zero values. The result of a population count operation would yield a 0 if and only if half of the bits were zero and another half 1. This behavior is not only impractical to implement, but it is also impossible in case of odd input dimension size for the binarization axis. Furthermore, input padding in the XGEMM is twofold. Binarization pads input matrices with additional zeroed out columns if the number of columns is not multiple of binarization bit-width (32 in case of XGEMM). Second padding happens during the tiling as described for the general case. In both cases, the matrices are padded with 0 bits or integers.

The same amount of zero bits pads both input matrices. These zero bits are XNOR-ed which results in the same increase of every element in the resulting matrix ($0 \; xnor \; 0 = 1$). That means that each element of the resulting matrix needs to be adjusted proportionally to the added bit (B) and tile padding (T).

$$B = 31 - (Cols_A - 1) \bmod 32 \tag{5.5}$$

$$T = TileSize - 1 - \big(((Cols_A + B) \gg 5) - 1\big) \bmod TileSize \tag{5.6}$$

$$Padding = B + T \ll 5 \tag{5.7}$$

Symbols $\ll$ and $\gg$ mark left and right bitshifts.

CUDA kernel

XGEMM CUDA kernel organizes threads into blocks of 32×32 to perform tiling. In addition to the traditional inputs, it receives the total padding value which it uses to reduce the final result. The kernel uses two tiled-size static shared memory buffers for tiles from both inputs. Threads operate in three phases. In the first phase, they fetch data from the global memory into the shared memory. If the global memory address is invalid, threads fill matching shared memory location with 0 values (tile padding). Threads synchronize after the load and proceed to phase 2.

Second and third phase utilize additional optimization of the XNOR vector product that reduces the total number of instructions by approximately $3N$ where N is the size of tile dimension. This optimization breaks the computation into two parts. Instead of performing the full XNOR dot product (see 4.1), in the first part only the xor and population count are used to accumulate the value, which represents the total number of 0 bits. That removes N bitshifts, not operations and subtractions. Finally, a thread has to subtract the doubled number of zeroes from the total number of processed bits which equates to $32 \times N \times k$, where k denotes number of tiles along the binarization axis.

In the second phase, all threads perform vector dot product of the corresponding rows and columns taken from the shared memory tiles and store the result into a register. After the thread warp processes all the tiles it adjusts the result according to the computation optimization and the padding, and stores it to the output memory in the third phase.

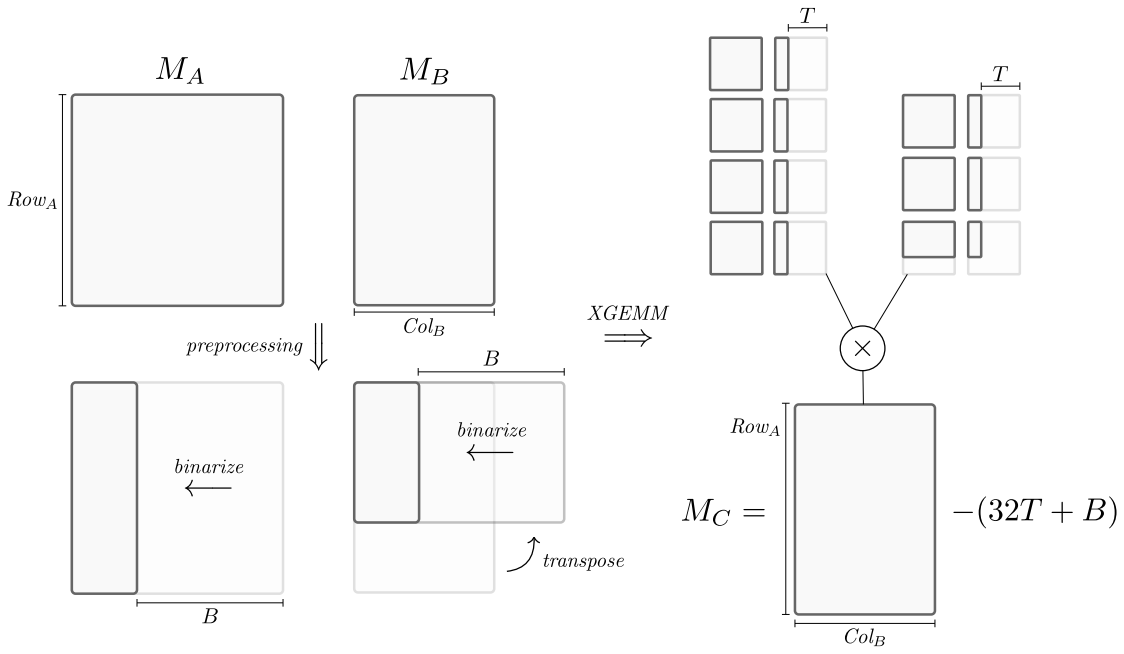


Figure 5.5: Visualisation of XGEMM operation implementation. The left-hand side shows input preprocessing with binarization operators. Right-hand side show XGEMM tiled computation where $\times$ symbol represents matrix multiplication using only xnor and population count instructions.

The implemented kernel does not take advantage of prefetching and loop unrolling.

5.3.3 XGEMM benchmark

Figure 5.6 shows the result of XGEMM implementation benchmark that compares speedup compared to the TensorFlow matmul operator which uses cuBLAS GEMM. Benchmark was executed on the NVIDIA GeForce 940MX GPU with 2GB memory (full device details are in Appendix E). Benchmark computed matrix multiplication of matrix A of size N rows by M columns and matrix B of size M rows by N columns. N is on the vertical axis as M is on the horizontal axis of the graph. White and red parts of the graph represent none or negative speedup while the green parts represent a real speedup.

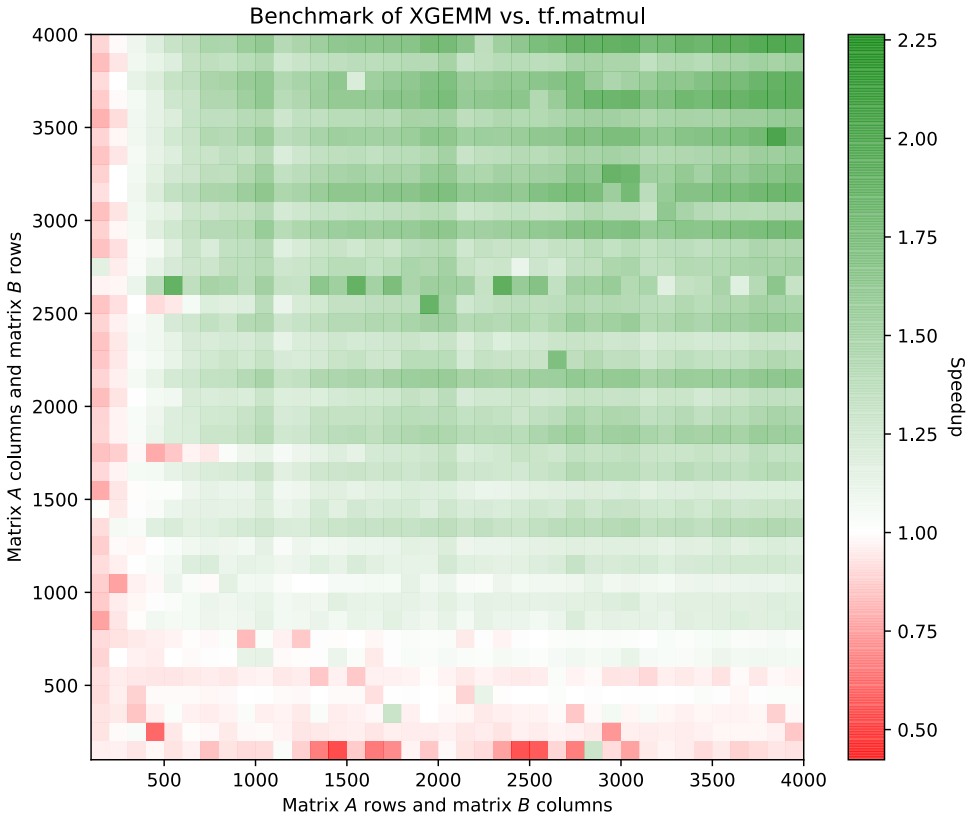


Figure 5.6: Dimensions of inputs are in range $\langle 100, 4000 \rangle$. Benchmark was executed for each combination of inputs in the range with step 100. Benchmark ran for 10-40 iterations for each combination of inputs and methods.

Even though there is symmetry in the input matrices, the graph is asymmetrical, showing steadily worse performance for low N. Binarization is the cause of this asymmetry. Inputs are binarized along the M axis. Therefore, the amount of time spent in binarization kernels raises quadratically with N. Speedup of the XGEMM overtakes overhead of the binarization with N larger than approximately 600. Speedup increases with the size of inputs reaching over 250% with 4000×4000 inputs.

Rastegari et al. proposed 58x faster convolution operations [34]. XGEMM achieves only a fraction of this proposition, and under specific conditions, the operation is even slower

than a full precision GEMM. There are two primary reasons for the discrepancy between theory and practice.

1. Theoretical work does not take into account the overhead related to binarization and latency introduced by the dense memory access.

2. Paper assumes an equal level of the implementation optimization for the full-precision and binary kernels. TensorFlow's cuBLAS GEMM is scrutinizingly optimized while the XGEMM implementation uses only the basic optimization techniques.

There is no doubt that the implementation can be drastically improved. The fact that the XGEMM implementation achieves speedup on reasonably sized inputs is extremely promising.

Chapter 6

MNIST classification with XGEMM

The previous chapter describes the concept and implementation of XGEMM. This chapter injects XGEMM into a neural network classification of the MNIST database and measures the achieved speedup and change in the classification accuracy. Implemented architecture is a simple multi-layer perceptron which utilizes novel methods for training of binarized networks suggested by Tang et al. in [44].

6.1 MNIST database

MNIST by Yann LeCun et al. [27] is a database of handwritten digits designed as an introductory machine-learning and pattern matching dataset. It consists of a training set with 60,000 examples and a test set of 10,000 examples. The digits are normalized by their size and centered in a 28 by 28 pixels images. Researchers use MNIST dataset for benchmarking due to its simplicity and ease of use.

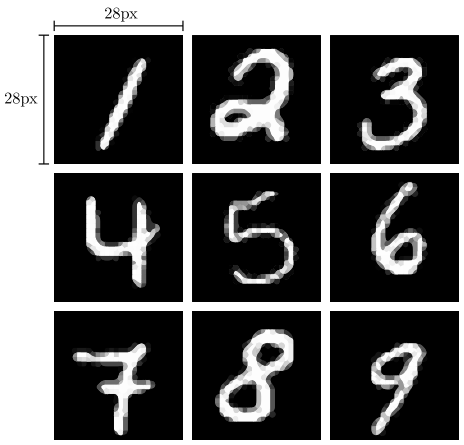


Figure 6.1: Sample preview of MNIST database images.

The training set contains samples from approximately 250 writers. Set of writers of the test set is disjoint from the set of training set writers. Current state-of-the-art classifiers perform better classification than human readers that achieve an average accuracy of 98.29%.

6.2 Full-precision network architecture

Implemented MNIST classifier consists of 4 fully-connected layers (see 2.1.2). This type of architecture is often referred to as a multi-layer perceptron (MLP). All layers except for the last one have the same number of neurons. First 3 layers use the rectified linear unit (ReLU) activation function. The last layer has 10 neurons and performs affine transformation with softmax activation to produce probabilities for all 10 classes. Figure 6.2 visualizes the model.

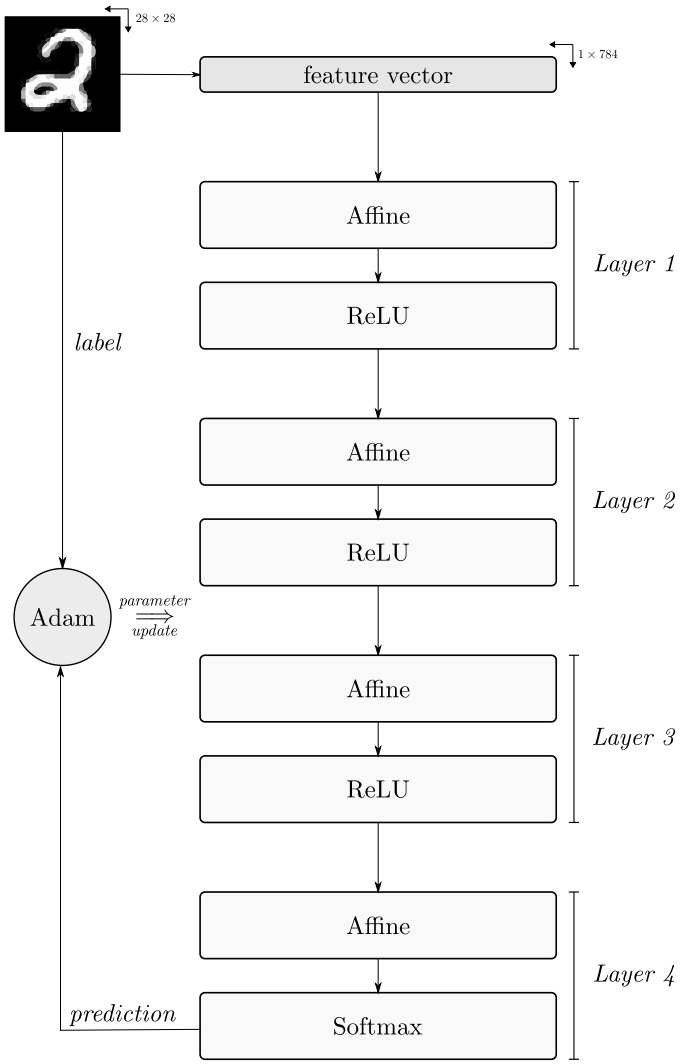


Figure 6.2: Full-precision network architecture.

The input image is flattened into a feature vector of 784 elements and fed into the network layer. Inference process produces predictions which are used to calculate network loss via the cross-entropy function. Adam optimization algorithm which combines advantages of AdaGrad and RMSProp algorithms [23] uses computed parameter gradients to calculate parameter updates.

MLP is not an ideal architecture for image recognition. However, it is the best choice for easy acceleration with XGEMM. Convolutional networks (see 2.1.2) are better suited for visual tasks and achieve better results with less effort. The following section defines a modified architecture utilizing XNN principles.

6.3 Binarized network model

Binarization of the full-precision model requires two notable adjustments.

1. Replace ReLU activation with binary activation (signum).
2. Quantization of weights before the affine transformation via XGEMM.

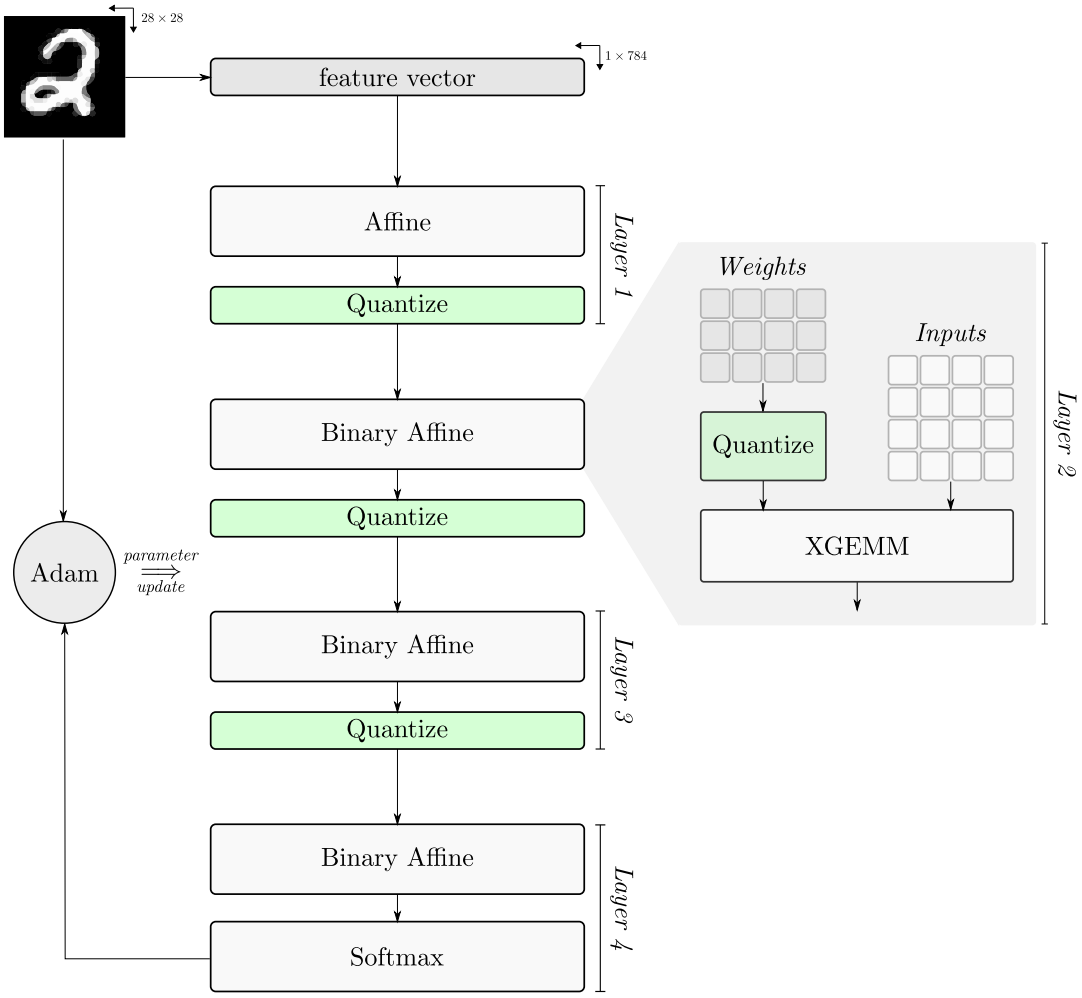


Figure 6.3: Binarized network architecture.

Figure 6.3 shows the change in network architecture. Green blocks labeled *Quantize* represent binary activations. *Binary Affine* is an affine transformation with binarized weights and XGEMM.

The focus of the implementation is not only on the acceleration but as well on a good accuracy while achieving the speedup. Research by Tang et al. shows that it is advantageous to keep full precision in the first layer to keep the higher quality of derived knowledge of the input features in later layers [44]. Empirical evidence supports implications of the assumption on the accuracy. Input feature vector is not sizeable enough to achieve significant speedup by applying XGEMM. Therefore it is fitting to omit binarization in the first layer.

6.4 Implementation and experiments

Implementation of full-precision MLP matches the architecture described in section 6.2 precisely as described. Binarized network code utilizes few critical features of implemented XGEMM. Firstly the XGEMM has binary activation and quantization embedded within, meaning calling signum to quantize activations and weights is a redundant computation. This feature shrinks the model into more compact version shown in figure 6.4.

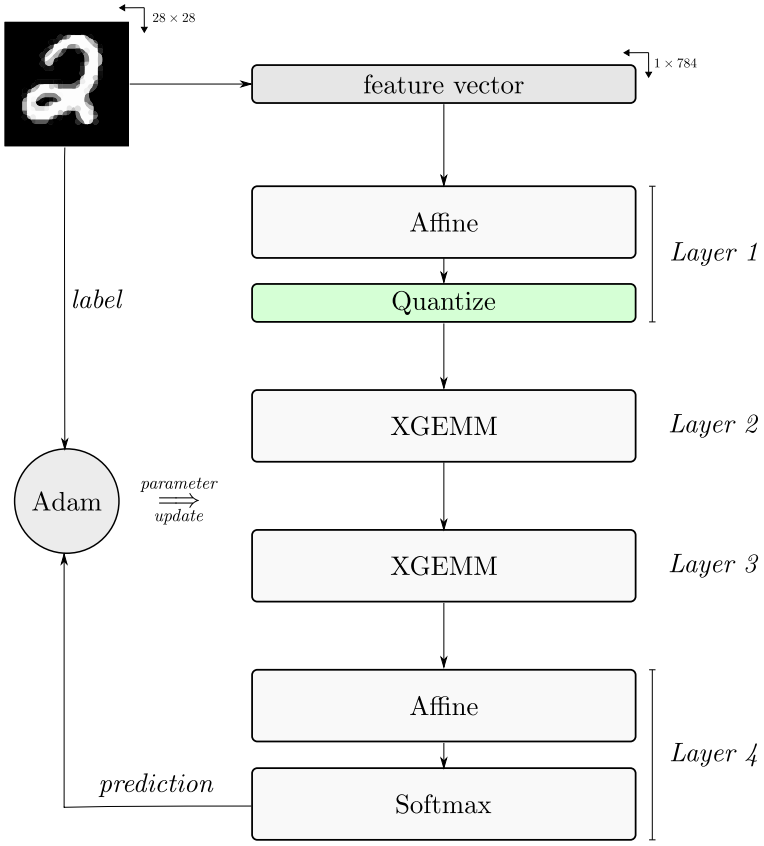


Figure 6.4: Compressed version of the binarized model.

An essential part of the implementation is XGEMM gradient computation. Gradients for custom operators can be defined directly in python with TensorFlow `RegisterGradient` decorator (see 5.1.1). Input is the gradient with respect to the output of the XGEMM (∇G) and both input matrices (A, B). Then gradients are computed according to equations 6.1.

Gradient computations are not quantized which increases computation time.

$$\begin{aligned} \nabla A &= \nabla G \times B^T \\ \nabla B &= A^T \times \nabla G \end{aligned} \tag{6.1}$$

The implemented binarized model also omits binarization in the output layer. Output width is only 10, which is too low to achieve any speedup through XGEMM.

6.4.1 MNIST benchmark

Most big models suffer from long training time. Mini-batch training reduces the training time significantly by reducing the number of gradient computations and parameter updates linearly with the batch size. Batch size determines the size of the input matrix of each network layer. Therefore it is an attribute of underlying matrix multiplications. Another attribute is the size of layers interpreted as the number of neurons.

Training in real applications has several options concerning when it should end. Typically the cross-validation method is used to terminate training process before the network overfits the training dataset or to tweak parameters such as learning rate or dropout probability. For the benchmarking purposes, the learning rate is constant $\eta = 0.005$ and number of training steps is constant relative to the input size. A training step refers to the processing of a single mini-batch. Table 6.1 shows the relation between the number of training steps and input sizes.

Input size	Steps
≤ 1000	1000
≤ 2000	600
> 2000	300

Table 6.1: The relation between the maximum of batch size and layer size, and the number of steps. A smaller number of steps decreases overall runtime.

Both models were tested on an NVIDIA GeForce 940MX GPU (see Appendix E) on operating system Fedora 27 with CUDA v9.1, cuDNN v7.0.5 and TensorFlow v1.5.0-rc0. Batch size ranged from 512 to 4096 increasing with step 512 and with the number of neurons in each hidden layer ranging from 1000 up to 4200 increasing by 400. The total amount of runs equals 162 (81 for each network).

Figure 6.5 shows the speedup achieved by binarization with XGEMM on the network. Speedup gradually increases with the layers size. The rate of increase is lower than for an isolated XGEMM (see 5.3.3). The leading cause is that the compressed binarized model is much more complicated than simple matrix multiplication. XGEMM layers are only half of the network and gradient computation, which is not quantized, consumes a significant amount of total training time. The general conclusion of the speedup benchmark is that increasing layer and batch size increase the achieved speedup.

Maximal speedup is by $\approx 25\%$ for layer size ≈ 4000. That might not seem like much, but with more fully-connected layers it should potentially reach the speedup of pure XGEMM at about 100%. A good example where this would help is a triphone classification of frames in speech recognition. Karel's implementation in Kaldi [33] uses 7 fully-connected layers with 3370 neurons classifying inputs into 1936 categories. Training time of such networks can be

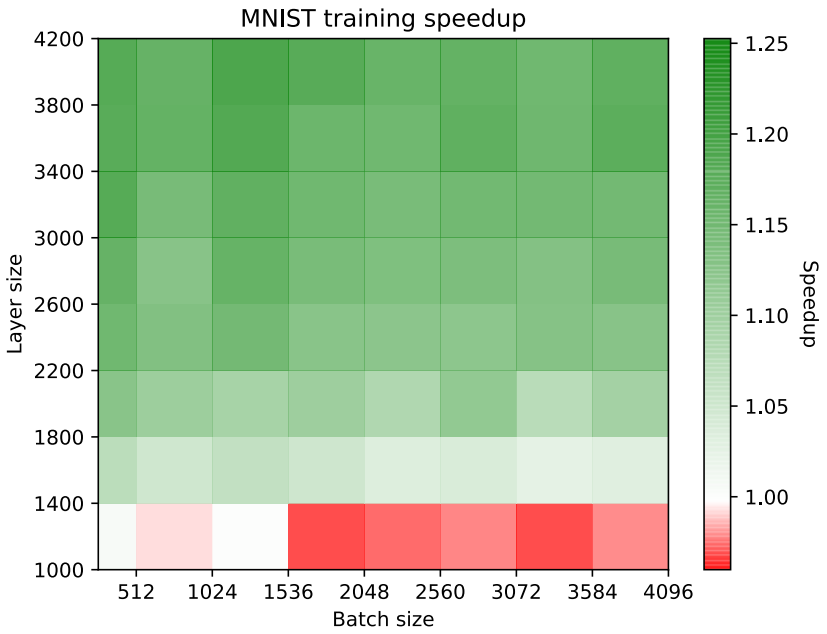


Figure 6.5: Speedup of the binarized network with XGEMM on MNIST classification.

several weeks depending on learning rate and speed of convergence. Reducing this amount to a half and significantly increasing the speed of trained network inference is excellent if binarized network maintains the accuracy.

Benchmark provides few insights concerning network accuracy. Table 6.2 shows statistical data gathered from all benchmark runs. On average the XGEMM network achieved $\approx 5.5\%$ drop inaccuracy with higher variance. Note that binarized network used the same hyperparameters as the full precision network. The question of how to train binarized models is the subject of several studies, which are surveyed in the next chapter.

Model	Max [%]	Min [%]	Mean [%]	Standard deviation [%]
Full-precision	97.02	94.57	96.01	±0.56
Binary XGEMM	91.91	87.68	90.49	±0.93

Table 6.2: Comparison of full-precision and binarized network models.

Chapter 7

Comparison of binarized models

Chapter 4 described several binarization techniques, most importantly the XNOR-net. This chapter briefly describes few additional methods utilizing binarization, namely Binarized-Neural-Network (BNN) by Courbariaux et al. [7] and the DoReFa-net by Zhou et al. [49]. Application of these methods into practice is still a complicated task because of a sudden drop in accuracy on large-scale datasets. Section reference of this chapter draws from research by Tang et al. [44] and shows several successful methods of improving binarized models.

7.1 Binarized-Neural-Network

BNN is built on top of the XNN principles and introduces an improved method of training networks with binary weights and activations, and when computing the parameters gradients during the training. Authors created an implementation of xnor-based matrix multiplication. However, this implementation is not a GEMM and works only for matrices with dimensions that are multiples of 512, and only for single-precision inputs and output.

Neural networks use regularization to prevent overfitting and achieve better generalization. Adding noise to network parameters provides a form of regularization. The best-known form of adding noise is the dropout method [40], which randomly sets a percentage of weights to zero when computing the gradients. Full-precision networks often incorporate dropout layers into their architecture. Binarization of weights and activations acts like a dropout. BNNs also utilize batch normalization on activations to achieve zero mean and unit variance.

BNNs layers are composed of a sequence of stochastic or deterministic binarization (4.1.1), affine transformation with binarized activation of the previous layer, and batch normalization. Gradients are computed with respect to non-binarized activations.

7.2 DoReFa-net

DoReFa-net (DRF) is a training method of quantized models with low bit-width weights, activations and parameter gradients. This approach does not restrict parameters to be binary. However, authors used 1-bit weights in their experiments and achieved accuracy comparable to full-precision models. Quantization of weights in DRFs is based on the XNN principle. Instead of using channel-wise scaling factors, DRF selects a constant scalar to scale all filters. That allows to the exploitation of binary convolutional kernels during the

backpropagation. Quantization of activations is a straightforward k-bit representation of each activation, where k is quantization bit-width.

7.3 Effective training strategies for binarized networks

Paper *How to Train a Compact Binary Neural Network with High Accuracy?* [44] provides insightful analysis of mentioned binarized models and based on careful observations proposes several strategies for enhanced training of binarized models achieving better compression rate and inference accuracy. Binarized models have unique features that are not observed in full-precision models. Previous works focused on achieving the best approximation of original models not paying attention to the underlying nature of binarized networks. Acknowledging these features allows devising better strategies for creating architectures of binarized models and their training.

Low learning rate

Selecting adequate learning rate is a key to successful training. Convolutional layers are commonly trained with the learning rate of approximately 0.01. Behavioral analysis of binarized networks showed high fluctuations in network accuracy leading to lower accuracy. Higher learning rate causes more frequent sign changes of weights introducing lower stability during training. Empirical evidence shows that the training process stabilizes with a lower learning rate. The learning rate can be as low as 0.0001.

PReLU activation function

Parametric rectified linear unit (PReLU) allows to omit the activation scale factors used in XNN and move weight scale factors into the activation function. This modification simplifies complicated convolution operation in XNN (see 4.2.3) and allows it to be carried out only by the XGEMM. When applied to BNN, this method further increases achieved accuracy.

Regularization

As mentioned previously, regularization increases network generalization capacity and helps to prevent overfitting. Ridge regression [20] (L_2 regularization) is standard practice in deep neural networks. Binarized models provide the best approximations when full-precision parameters have values close to ± 1. To achieve this Tang et al. propose an alternative regularization term that pilots weights to ± 1:

$$J(W,b) = L(W,b) + \lambda \sum_{l=1}^{L} \sum_{i=1}^{N_l} \sum_{j=1}^{M_l} \left(1 - (W_{ij}^l)^2\right) \tag{7.1}$$

N_l and M_l in the equation 7.1 denote the dimensions of the weight matrix in the l-th layer. $L(W,b)$ is loss associated with the network for input batch. Rest of the equation is the proposed regularization term parametrized by λ.

Scaling the last layer

Binarization of the last layer proved to have a significant adverse impact on network accuracy and compression rate. Outputs of the last layer in full-precision classifiers correspond

to the probabilistic distribution of prediction among the classes. When both inputs and weights of the last layer are binarized, the domain of output activation decreases significantly relative to the second to last layer size. The low range of output values does not befit the softmax function which is usually the last step of classification inference. Introducing simple scale layer defined by a learnable scalar scale parameter in between the binarized last layer and the softmax activation function improves compression rate and reduces accuracy drop caused by binarization.

Multiple binarizations

Mentioned binarized models use signum function to binarize both weights as activations directly. Tang et al. propose iterative binarization process that computes successive binarizations based on the residual approximation error. This strategy results in significantly better accuracy but at the cost of multiple computations of the affine transformation. To achieve any speedup this method requires exceptionally well optimized XGEMM implementation.

Expanding lower layers

Han et al. [17] showed lesser parameter redundancy in lower layers of neural networks. Binarization significantly reduces network representation capacity which can be regained by expanding network layers. This process needs to be done carefully to keep high compression rate and speedup. Therefore it is advisable to expand only the lower layers.

7.4 MNIST, SVHN and CIFAR-10 classification accuracy

Following tables compare binarization methods on various architectures on MNIST (see 6.1), SVHN (the street view house numbers dataset) and CIFAR-10 [24] datasets. The last row of each table refers to the state-of-the-art full-precision solution. All other full-precision solutions are in highlighted rows. Accuracy measurements were taken over from the original studies.

Method	**Architecture**	**Error Rate [%]**
BinaryConnect	MLP	1.29
BNN	3-layer MLP	0.96
XNN	LeNet-5	0.77
Full-precision	LeNet-5	0.80
Full-precision	2-layer MLP with DropConnect [46]	0.21

Table 7.1: Comparison network binarization methods on the MNIST dataset.

Method	Architecture	Error Rate [%]
BinaryConnect	CNN inspired by VGG [37]	2.30
BNN	CNN inspired by VGG	2.53
Full-precision	Gated CNN [28]	1.69

Table 7.2: Comparison network binarization methods on the SVHN dataset.

Method	Architecture	Error Rate [%]
BinaryConnect	CNN inspired by VGG	8.27
BNN	CNN inspired by VGG	9.90
XNN	Network-in-Network (NIN) [29]	13.72
Full-precision	NIN	8.80
Full-precision	CNN with fractional max-pooling [14]	3.47

Table 7.3: Comparison network binarization methods on the CIFAR-10 dataset.

MNIST, SVHN, and CIFAR-10 visual recognition tasks are relatively small-scale and easy to solve. Tables 7.1, 7.2, and 7.3 reflect that not only by showing low error rates on the state-of-the-art solutions. Binarized models achieve the same accuracy on MNIST and comparable accuracy on SVHN and CIFAR-10. These results are auspicious. However, binarized models perform noticeably worse on large-scale datasets, such as ImageNet as shown in the next section.

7.5 ImageNet classification accuracy

The ImageNet dataset consists of over 14 million annotated images with approximately 20 thousand categories. It is considered to be a large-scale dataset. Accuracy on the ImageNet is measured on a top-1 and top-5 scale. Top-N accuracy is generalized accuracy measurement scale for classification tasks. Neural network prediction is considered correct if the target class is in the top N predictions. For example, all MNIST classifiers have 100% top-10 accuracy. Top-1 accuracy is the default accuracy measurement.

Table 7.4 compares binarization methods on the ImageNet dataset. BinaryConnect and BWN methods that binarize only the weights show better results than fully binarized BNN or XNN. However, application of efficient training strategies on BNN improved its performance with the AlexNet architecture by almost 20% for both Top-1 and Top-5 accuracy. BWN achieves impressive result when applied to GoogLenet architecture, dropping accuracy only by approximately 6%. However, based on the ResNet-18 benchmark, XNN would perform with much lower accuracy. Unfortunately, no research applied binarized training optimization on these well-performing architectures.

Method	**Architecture**	**Top-1** [%]	**Top-5** [%]
BNN	AlexNet (Appendix C)	27.90	50.42
BinaryConnect	AlexNet	35.40	61.00
XNN	AlexNet	44.20	69.50
DoReFa-net	AlexNet (2-bit activations)	49.80	–
Optimized BNN	AlexNet (2-bit activations)	46.60	71.10
Optimized BNN	NIN (2-bit activations)	51.40	75.60
Full-precision	AlexNet	56.60	80.20
XNN	ResNet-18 [18]	51.20	73.20
BWN (see 4.2.1)	ResNet-18	60.80	83.00
Full-precision	ResNet-18	69.30	89.20
BWN	GoogLenet [42]	65.50	86.10
Full-precision	GoogLenet	71.30	90.00
Full-precision	Inception ResNet-v2 [41]	83.50	96.90

Table 7.4: Comparison network binarization methods on the ImageNet dataset.

Chapter 8

Conclusion

This thesis contributes in two ways. Firstly it comprehensibly summarizes neural network acceleration techniques with stress on network binarization. Binarization allows replacing expensive floating point multiplications with faster bitwise operators, and process neural network computations in a SIMD fashion. Researchers developed several methods that maximize network speedup while minimizing the loss of accuracy. Network binarization has achieved palpable advancements in the field over the course of past 3 years. However, it is still relatively unknown among the machine-learning community, with to my knowledge no source summarizing the plethora of binarization studies up until now.

The second more practical contribution is in the form of a TensorFlow operator suite which allows developing binarized networks in one of the most popular machine-learning frameworks of today. These operators are implemented in templated CUDA and C++ code runnable on CUDA-enabled GPUs. Speedup benchmark revealed 2.5 faster execution time on reasonably sized inputs. This result is much lower than the theoretical supposition, showing that there is still much space for an improvement of the implementation.

Theoretical base and TensorFlow implementation were merged in a binarized implementation of MNIST classifier and compared to the full-precision counterpart. Binarized and full-precision architectures were trained and compared with different batch and hidden layer sizes. The maximal achieved speedup was only by approximately 25% which would grow with network size. These results imply that the current implementation is not ready for the commercial use. However, the fact that it achieved any speedup at all is a good sign moving forward. Various binarized techniques are compared by the achieved accuracy on different architectures and tasks in the last chapter, showing promising results. Recent advancements keep relatively good accuracy even on large-scale datasets.

Binarization faces difficult challenges on both theoretical and practical fronts. Studies showed that binarized networks have unique nature and require a specialized approach to modeling and training. The future work should focus on exploiting these features in algorithms tailored for binarized models. In addition to that, the implemented operator suite for the TensorFlow is small and not optimized to the maximum. The smallest step forward would be to implement binarized convolution utilizing the existing operators or further accelerating them, which would require extensive CUDA programming skills.

Bibliography

[1] Ba, L. J.; Caurana, R.: Do Deep Nets Really Need to be Deep? *CoRR*. vol. abs/1312.6184. 2013.
Retrieved from: http://arxiv.org/abs/1312.6184

[2] Canziani, A.; Paszke, A.; Culurciello, E.: An Analysis of Deep Neural Network Models for Practical Applications. *CoRR*. vol. abs/1605.07678. 2016. 1605.07678.
Retrieved from: http://arxiv.org/abs/1605.07678

[3] Chen, W.; Wilson, J. T.; Tyree, S.; et al.: Compressing Neural Networks with the Hashing Trick. *CoRR*. vol. abs/1504.04788. 2015.
Retrieved from: http://arxiv.org/abs/1504.04788

[4] Cheng, Y.; Wang, D.; Zhou, P.; et al.: A Survey of Model Compression and Acceleration for Deep Neural Networks. 10 2017.
Retrieved from: https://arxiv.org/abs/1710.09282

[5] Choi, Y.; El-Khamy, M.; Lee, J.: Towards the Limit of Network Quantization. *CoRR*. vol. abs/1612.01543. 2016.
Retrieved from: http://arxiv.org/abs/1612.01543

[6] Cohen, T. S.; Welling, M.: Group Equivariant Convolutional Networks. *CoRR*. vol. abs/1602.07576. 2016.
Retrieved from: http://arxiv.org/abs/1602.07576

[7] Courbariaux, M.; Bengio, Y.: BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*. vol. abs/1602.02830. 2016.

[8] Courbariaux, M.; Bengio, Y.; David, J.: BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *CoRR*. vol. abs/1511.00363. 2015.
Retrieved from: http://arxiv.org/abs/1511.00363

[9] Cun, Y. L.; Denker, J. S.; Solla, S. A.: Optimal Brain Damage. In *Advances in Neural Information Processing Systems 2*, edited by D. S. Touretzky. San Francisco, CA: Morgan Kaufmann. 1990. pp. 598–605.

[10] Diamos, G.; Sengupta, S.; Catanzaro, B.; et al.: Persistent RNNs: Stashing Recurrent Weights On-Chip. In *ICML*. 2016.
Retrieved from: http://proceedings.mlr.press/v48/diamos16.pdf

[11] Dieleman, S.; Fauw, J. D.; Kavukcuoglu, K.: Exploiting Cyclic Symmetry in Convolutional Neural Networks. *CoRR*. vol. abs/1602.02660. 2016.
Retrieved from: http://arxiv.org/abs/1602.02660

[12] Du, K.-L.; Swamy, M. N. S.: *Neural Networks in a Softcomputing Framework*. Springer Publishing Company. 2006. ISBN 1-84628-302-7.

[13] Gong, Y.; Liu, L.; Yang, M.; et al.: Compressing Deep Convolutional Networks using Vector Quantization. *CoRR*. vol. abs/1412.6115. 2014.
Retrieved from: http://arxiv.org/abs/1412.6115

[14] Graham, B.: Fractional Max-Pooling. *CoRR*. vol. abs/1412.6071. 2014. 1412.6071.
Retrieved from: http://arxiv.org/abs/1412.6071

[15] Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; et al.: Deep Learning with Limited Numerical Precision. *CoRR*. vol. abs/1502.02551. 2015.
Retrieved from: http://arxiv.org/abs/1502.02551

[16] Han, S.; Liu, X.; Mao, H.; et al.: EIE: Efficient Inference Engine on Compressed Deep Neural Network. *CoRR*. vol. abs/1602.01528. 2016.
Retrieved from: http://arxiv.org/abs/1602.01528

[17] Han, S.; Mao, H.; Dally, W. J.: Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*. vol. abs/1510.00149. 2015.
Retrieved from: http://arxiv.org/abs/1510.00149

[18] He, K.; Zhang, X.; Ren, S.; et al.: Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016: pp. 770–778.
Retrieved from: https://arxiv.org/abs/1512.03385

[19] Hinton, G.; Vinyals, O.; Dean, J.: Distilling the knowledge in a neural network. *CoRR*. vol. abs/1503.02531. 2015.
Retrieved from: https://arxiv.org/abs/1503.02531

[20] Hoerl, A. E.; Kennard, R. W.: Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*. vol. 12, no. 1. 1970: pp. 55–67. doi:10.1080/00401706.1970.10488634.
https://amstat.tandfonline.com/doi/pdf/10.1080/00401706.1970.10488634.
Retrieved from:
https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634

[21] Ioffe, S.; Szegedy, C.: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. vol. abs/1502.03167. 2015.
Retrieved from: http://arxiv.org/abs/1502.03167

[22] Jaderberg, M.; Vedaldi, A.; Zisserman, A.: Speeding up Convolutional Neural Networks with Low Rank Expansions. *CoRR*. vol. abs/1405.3866. 2014.
Retrieved from: http://arxiv.org/abs/1405.3866

[23] Kingma, D. P.; Ba, J.: Adam: A Method for Stochastic Optimization. *CoRR*. vol. abs/1412.6980. 2014. 1412.6980.
Retrieved from: http://arxiv.org/abs/1412.6980

[24] Krizhevsky, A.: Learning multiple layers of features from tiny images. Technical report. 2009.
Retrieved from:
https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[25] Krizhevsky, A.; Sutskever, I.; Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, edited by F. Pereira; C. J. C. Burges; L. Bottou; K. Q. Weinberger. Curran Associates, Inc.. 2012. pp. 1097–1105.
Retrieved from: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[26] Lebedev, V.; Ganin, Y.; Rakhuba, M.; et al.: Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. *CoRR*. vol. abs/1412.6553. 2014.
Retrieved from: http://arxiv.org/abs/1412.6553

[27] LeCun, Y.; Cortes, C.; Burges, C. J.: The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/. accessed: 2018-05-13.

[28] Lee, C.-Y.; Gallagher, P. W.; Tu, Z.: Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, *Proceedings of Machine Learning Research*, vol. 51, edited by A. Gretton; C. C. Robert. Cadiz, Spain: PMLR. 09–11 May 2016. pp. 464–472.
Retrieved from: http://proceedings.mlr.press/v51/lee16a.html

[29] Lin, M.; Chen, Q.; Yan, S.: Network In Network. *CoRR*. vol. abs/1312.4400. 2013. 1312.4400.
Retrieved from: http://arxiv.org/abs/1312.4400

[30] Lin, Z.; Courbariaux, M.; Memisevic, R.; et al.: Neural Networks with Few Multiplications. *CoRR*. vol. abs/1510.03009. 2015.
Retrieved from: http://arxiv.org/abs/1510.03009

[31] McCulloch, W. S.; Pitts, W.: *A logical calculus of the ideas immanent in nervous activity*. The bulletin of mathematical biophysics. 1943.

[32] Merolla, P.; Appuswamy, R.; Arthur, J. V.; et al.: Deep neural networks are robust to weight binarization and other non-linear distortions. *CoRR*. vol. abs/1606.01981. 2016.
Retrieved from: http://arxiv.org/abs/1606.01981

[33] Povey, D.; Ghoshal, A.; Boulianne, G.; et al.: The Kaldi Speech Recognition Toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society. December 2011. iEEE Catalog No.: CFP11SRW-USB.

[34] Rastegari, M.; Ordonez, V.; Redmon, J.; et al.: XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR*. vol. abs/1603.05279. 2016.
Retrieved from: http://arxiv.org/abs/1603.05279

[35] Romero, A.; Ballas, N.; Kahou, S. E.; et al.: FitNets: Hints for Thin Deep Nets. *CoRR*. vol. abs/1412.6550. 2014.
Retrieved from: http://arxiv.org/abs/1412.6550

[36] Ruetsch, G.; Fatica, M.: *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. first edition. 2013. ISBN 0124169708, 9780124169708.

[37] Simonyan, K.; Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. vol. abs/1409.1556. 2014. 1409.1556.
Retrieved from: http://arxiv.org/abs/1409.1556

[38] Springenberg, J. T.; Dosovitskiy, A.; Brox, T.; et al.: Striving for Simplicity: The All Convolutional Net. *CoRR*. vol. abs/1412.6806. 2014.
Retrieved from: https://arxiv.org/abs/1412.6806

[39] Srinivas, S.; Babu, R. V.: Data-free parameter pruning for Deep Neural Networks. *CoRR*. vol. abs/1507.06149. 2015.
Retrieved from: http://arxiv.org/abs/1507.06149

[40] Srivastava, N.; Hinton, G.; Krizhevsky, A.; et al.: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. vol. 15. 2014: pp. 1929–1958.
Retrieved from: http://jmlr.org/papers/v15/srivastava14a.html

[41] Szegedy, C.; Ioffe, S.; Vanhoucke, V.: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *CoRR*. vol. abs/1602.07261. 2016. 1602.07261.
Retrieved from: http://arxiv.org/abs/1602.07261

[42] Szegedy, C.; Liu, W.; Jia, Y.; et al.: Going Deeper with Convolutions. *CoRR*. vol. abs/1409.4842. 2014. 1409.4842.
Retrieved from: http://arxiv.org/abs/1409.4842

[43] Tai, C.; Xiao, T.; Zhang, Y.; et al.: Convolutional neural networks with low-rank regularization. 2015.
Retrieved from: http://arxiv.org/abs/1511.06067

[44] Tang, W.; Hua, G.; Wang, L.: How to Train a Compact Binary Neural Network with High Accuracy? 2017.
Retrieved from:
https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14619

[45] Vanhoucke, V.; Senior, A.; Mao, M. Z.: Improving the speed of neural networks on CPUs. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[46] Wan, L.; Zeiler, M.; Zhang, S.; et al.: Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, vol. 28, edited by S. Dasgupta; D. McAllester. Atlanta, Georgia, USA: PMLR. 17–19 Jun 2013. pp. 1058–1066.
Retrieved from: http://proceedings.mlr.press/v28/wan13.html

[47] Wen, Z.; Yin, W.; Zhang, Y.: Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm. *Mathematical Programming Computation*. vol. 4, no. 4. Dec 2012: pp. 333–361. ISSN 1867-2957. doi:10.1007/s12532-012-0044-1.
Retrieved from: https://doi.org/10.1007/s12532-012-0044-1

[48] Yang, Z.; Moczulski, M.; Denil, M.; et al.: Deep Fried Convnets. *CoRR*. vol. abs/1412.7149. 2014.
Retrieved from: http://arxiv.org/abs/1412.7149

[49] Zhou, S.; Ni, Z.; Zhou, X.; et al.: DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR*. vol. abs/1606.06160. 2016. 1606.06160.
Retrieved from: http://arxiv.org/abs/1606.06160

Appendix A

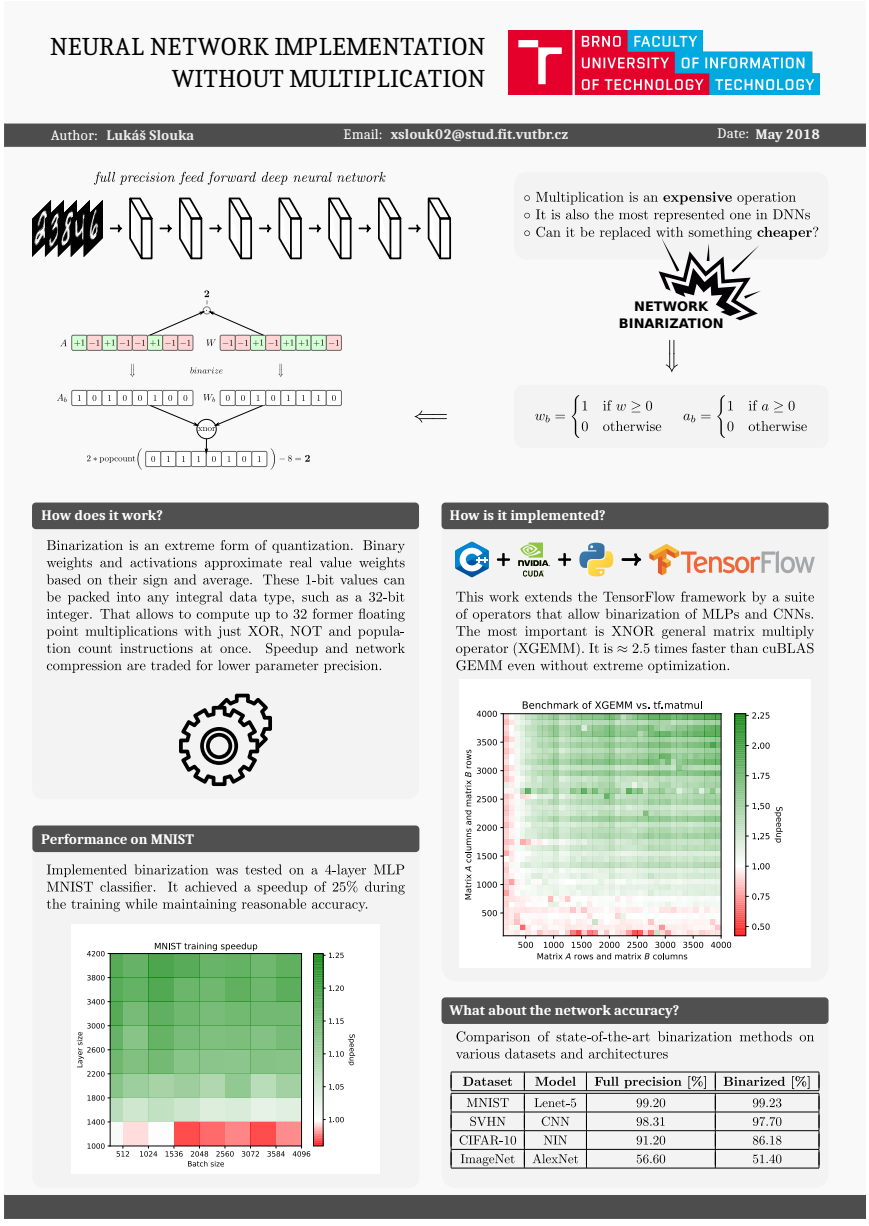
Memory media contents

```
/
├── thesis ...  thesis text
│   ├── figures ...  svg and pdf figures
│   ├── poster ...  poster sources
│   ├── *.tex ...  latex source files
│   └── Makefile
├── src ...  implementation
│   ├── benchmarks ...  python benchmark code
│   ├── graph ...  graphing utilities
│   ├── MNIST_data ...  the MNIST dataset
│   ├── models ...  network archtectures in TensorFlow
│   ├── operators ...  new operators backend implementation in C++ and CUDA
│   ├── op_interface ...  new operators frontend - python interface
│   ├── resources ...  image resources for README files
│   ├── results ...  benchmark and MNIST results
│   ├── tests ...  test source files for the implemented operators
│   ├── compare_mnist.sh ...  execution script for the MNIST comparison
│   ├── configuration.py
│   ├── run_benchmarks.py
│   ├── run_mnist.py ...  trains a MNIST model with given arguments
│   ├── run_tests.py ...  executes tests with given arguments
│   ├── README.md
│   └── LICENCE ...  MIT licence
├── dip.pdf ...  complete thesis in pdf
├── poster.pdf
└── video.mp4 ...  short video description of the work
```

Source files are also publicly available here https://github.com/LukasSlouka/TF_XNN.

Appendix B

Poster



Dataset	Model	Full precision [%]	Binarized [%]
MNIST	Lenet-5	99.20	99.23
SVHN	CNN	98.31	97.70
CIFAR-10	NIN	91.20	86.18
ImageNet	AlexNet	56.60	51.40

Appendix C

AlexNet

In 2012 the ImageNet challenge was won in a landslide by deep convolutional network now known as AlexNet. The goal was to train a network to classify 1.3 million high-resolution images into 1000 different classes. The developed network had 5×10^5 neurons and contained 5 convolutional layers. AlexNet had 6×10^7 learnable parameters [25].

The input of the first convolutional layer was an image of size $[227, 227, 3]$. Krizhevsky decided to use neurons with receptive field size $F = 11$, stride $S = 4$ without any zero padding. Depth of this layer was $N = 96$. By application of equations (2.1.2) resulting volume had size $[55, 55, 96]$. This means that the first layer of AlexNet has $55 \times 55 \times 96 = 290,400$ neurons, each with $11 \times 11 \times 3 = 363$ weights and 1 bias. That means that the number of parameters of this layer is over 10^8, which is more than total number of parameters mentioned before. AlexNet authors reduced this number down to $\sim 35,000$ with a parameter sharing strategy (see 3.1).

To put it in contrast, consider using fully-connected layer instead. Input vector size is $227 \times 227 \times 3 = 154,587$. That means that single fully-connected neuron would produce over 5-times more learnable parameters than used the first convolutional layer of AlexNet. Without parameter sharing, it would take less than 650 fully-connected neurons to produce the same amount of learnable parameters as a convolutional layer with $300,000$ neurons.

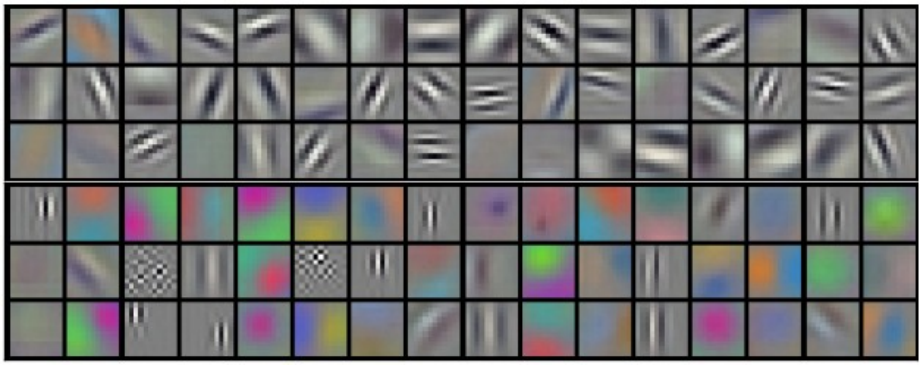


Figure C.1: All 96 filters learned by AlexNet. We can see that some filters detect edge patterns and other detect colored groups of pixels.

Appendix D

Deep-compression

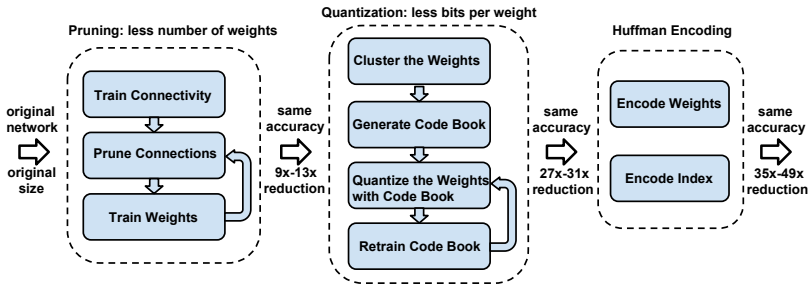


Figure D.1: Flow diagram of the three-stage deep compression algorithm that achieves up to 49 times better space complexity compared to the original network.

As shown in figure 2, the 3 stages of deep compression are pruning, quantization and Huffman encoding which is a classical form of lossless data compression. Any of these steps can serve as a standalone optimization.

Deep-compression algorithm builds on top of pruning method presented by Han et al. in 2015 [16] which is state-of-the-art pruning for convolutional models with no loss of accuracy. Pruning algorithm starts with regular network training followed by removing all weights with values lesser than a predefined threshold. After obtaining the connectivity, the network is retrained once again to learn final weights of selected sparse connections. Resulting sparse weights could be stored in efficient compressed sparse row/column (CSR/CSC) format.

Quantization algorithms in general compress networks by reducing the required number of bits per weight. To do so, the deep-compression algorithm uses k-means clustering of connection weight parameters ($n \gg k$ where n = the number of weights). Clusters of these weights have for sufficient k practically same value. Average values of clusters form vector called *centroid*, which represents shared values for clustered weights. During back-propagation update, all gradients of weights belonging to the same cluster are grouped, and SUM reduced, multiplied by the learning rate and subtracted from the centroid, resulting in *fine-tuned centroid*. Equation D.1 shows network compression rate, given k clusters, n weights and b number of bits required to represent each connection.

$$r = \frac{nb}{nlog_2(k) + kb} \tag{D.1}$$

Appendix E

NVIDIA GeForce 940MX GPU

Parameter	**Value**
CUDA Capability Major/Minor version number	5.0
Total amount of global memory	2004 MB (2,101,870,592 bytes)
Multiprocessors (3) × CUDA Cores/MP (192)	576 CUDA Cores
GPU Clock rate	1242 MHz (1.24 GHz)
Memory Clock rate	1001 MHz
Memory Bus Width	64-bit
L2 Cache Size	1 MB (1,048,576 bytes)
Max Texture Dimension Size 1D	$x \rightarrow 65536$
Max Texture Dimension Size 2D	$x \rightarrow 65536$, $y \rightarrow 65536$
Max Texture Dimension Size 3D	$x \rightarrow 4096$, $y \rightarrow 4096$, $z \rightarrow 4096$
Max Layered Texture Size (dim) x layers 1D	(16384) × 2048
Max Layered Texture Size (dim) x layers 2D	(16384,16384) × 2048
Total amount of constant memory	64 KB (65,536 bytes)
Total amount of shared memory per block	48 KB (49,152 bytes)
Total number of registers available per block	64 KB (65,536 bytes)
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Maximum sizes of each dimension of a block	1024 × 1024 × 64
Maximum sizes of each dimension of a grid	2,147,483,647 × 65,535 × 65,535
Maximum memory pitch	≈ 2 GB (2,147,483,647 bytes)
Texture alignment	512 bytes
Concurrent copy and kernel execution	Yes with 1 copy engine(s)
Run time limit on kernels	Yes
Integrated GPU sharing Host Memory	No
Support host page-locked memory mapping	Yes
Alignment requirement for Surfaces	Yes
Device has ECC support	Disabled
Device supports Unified Addressing (UVA)	Yes

Table E.1: A curated output of a CUDA `deviceQuery`.