Universität Potsdam

Institut für Informatik und Computational Science

Master's Thesis in Cognitive Systems: Language, Learning and Reasoning

Low-Precision Arithmetic for Convolutional Neural Network Inference

C. Clayton Violand

Universität Potsdam

Institut für Informatik und Computational Science

Master's Thesis in Cognitive Systems: Language, Learning and Reasoning

Low-Precision Arithmetic for Convolutional Neural Network Inference

Author: C. Clayton Violand

Advisor: Dr. Niels Landwehr

Add. Reader: Dr. Paul Prasse

Submission: 30.10.2017

I assure the single handed composition of this master's thesis only supported by declared resources.

Berlin, 19.08.2017

(C. Clayton Violand)

Abstract

It is well-known in principle that the full precision of computer hardware (e.g. 32- or 64-bit floats) is not really needed for neural networks. By design, they are robust against phenomena like small perturbations in data, stochastic activations of units (e.g. dropout) and weight randomization. Thus, there is a recent trend of moving to low-precision calculations in order to speed up inference in deep neural networks. In fact, recently designed GPUs feature an 8-bit integer mode specifically tailored for inference in deep neural networks. This would in theory run inference at four times the speed of 32-bit floats. However, there are few studies on exactly how low-precision calculations affect classification accuracy and speed at inference time. This work deals with the implementation of several neural network architectures in C++, the analysis of how well they hold up given a change in arithmetic precision, and the creation of a robust benchmarking tool equipped to carry out such analyses.

Keywords: convolutional neural networks, benchmarking, GEMM, integer arithmetic, image recognition

Inhaltsangabe

Es ist allgemein bekannt, dass die volle Genauigkeit der gegebenen Computerhardware (32bit oder 64bit) für neuronale Netzwerke nicht benötig wird. Durch ihr Design sind sie robust gegenüber kleinen Störungen in den Daten, stochastischer Aktivierungen von einzelnen Einheiten, wie z.B. durch Dropout, und gegen die zufällige Änderung von Gewichten. Hierdurch ergibt sich ein aktueller Trend zur Berechnung mittels niedrigerer Genauigkeit, um eine bessere Geschwindigkeit der Berechnungsoperationen zu erreichen. Aktuell hergestellte Grafikkarten stellen einen 8bit Integer Modus bereit, der speziell für die Verarbeitung innerhalb von neuronalen Netzen angepasst ist. Theoretisch würde dies zu einer vierfachen Geschwindigkeitssteigerung gegenüber der 32bit Fließkommavariante führen. Jedoch gibt es einige Studien, die genau aufzeigen, welchen Einfluss eine niedrigere Genauigkeit bei der Berechnung auf die Ergebnisse bei Klassifikation hat und wie die Laufzeit beeinflusst wird. In dieser Arbeit werden Implementierungen verschiedener neuronaler Netzwerkarchitekturen in C++ beleuchtet. Weiterhin wird analysiert, welche Änderung im Ergebnis durch die Verringerung der Genauigkeit entsteht. Außerdem wird eine stabile Umgebung entwickelt um Benchmarks sicher zu erheben.

Contents

List of Tables

List of Figures

List of Equations

Chapter 1

Introduction

Computations in convolutional neural networks (CNNs) require large matrix multiplication operations, with most of the heavy lifting being performed by the convolution layer. In the case of the well-known AlexNet architecture, 89% of CPU processing time is spent on convolutions [Ward 15]. As such, these networks can be costly and are slow at inference time. Different methods exist to implement convolution. The GEMM method is a popular method and involves only one large matrix multiplication operation per convolution layer. Still, this operation can be slow due to the size of the matrices and the intensive nature of matrix multiplication. One solution to this problem is to reduce the computation precision at the convolution layer, thus speeding up inference. However, this adjustment may also reduce performance accuracy. This paper deals with the trade-off between speed and performance accuracy as a result of performing integer multiplication for convolutions with GEMM, and the creation of a framework suitable for testing these performance differences.

All neural network computations involve "neurons", or nodes, with weights and biases that are learnable through training. In simple feed-forward neural networks such as single- or multi-layer perceptrons, the layers are "fully connected", meaning the weights in each layer are multiplied with *each* node of the input matrix, then all products in a particular layer are summed together. The output is then passed on to a non-linear transformation function and pushed forward in the pipeline of layers. The perceptron feed-forward neural network represents the most basic neural network, and was the first devised [Auer 08].

Images are large and don't work well with fully-connected perceptron neural network architectures. An image of small size, perhaps 28×28 pixels and three channels (RGB), will already require a network with $28 \times 28 \times 3 = 2352$ weights for each neuron. Such a network

would surely contain multiple neurons as well. Therefore, it's clear that fully-connected neural networks don't scale well to image data.

Convolutional neural networks are advantageous because they reduce the amount of parameters needed, and exploit "pixel neighborhoods". The convolution layer forms the foundation of convolutional neural networks. In such a layer, small weight matrices—or kernels—acting as a kind of filter, are multiplied with only a section of the input the size of the kernel itself. A predetermined number of these kernels iterate over the input in several matrix product calculations but nevertheless drastically reduce the amount of parameters, as each iteration over the input is met by the same kernel. In other words, the weight parameters are recycled. This duplication of parameters is possible due to the inherent nature of images. It can be said that a pattern that is useful in one section of the image might also be useful in another section of the image: this is what is meant by "exploitation of pixel neighborhoods". In this way, kernel weights are duplicated across product iterations over the input.

However, these networks are large. Recently, researchers have been moving towards quantization to enable faster inference times for these networks [Ward 16]. Quantization involves a translation of floating-point representations to integers, thus reducing computational resource needs. In theory, quantization could speed up inference by a factor of four. However, in practice, quantization is met with its own challenges. Firstly, each quantized convolution takes several parameters. Current techniques suggest a calculation of these parameters "offline" as much as possible. This work will show how important this offline calculation really is when it comes to reducing inference time. Secondly, quantization itself can slow down inference when not implemented in an optimal way. This, too, will be shown to be costly and a worthy candidate for optimization.

Many CNN architectures have been trained to address the image recognition problem, making great strides in reducing memory consumption and speed at both training and inference time. Some seminal networks include those born out of the Imagenet Large Scale Visual Recognition Challenge, or ILSVRC, trained on a popular dataset of the same name, drawing from a database of over ten million hand-annotated images [Deng 09]. The VGG family of models from Oxford University [Chat 14] are one example, as are Deep Residual Networks, winning first-place at the ILSVRC competition in 2015 [He 15]. Deep Residual Networks, or "ResNets", operate by intertwining layers in a non-linear fashion, and happen to benefit greatly from increased network depth. They have significantly reduced parameter sizes, however, and thus speed up inference. For example, while the 16- and 19-layer VGG neural networks require 15.3 and 19.6 billion FLOPS (floating-point operations)—or multiply-add operations—respectively, the 151 layer ResNet151 only

requires 11.3 billion FLOPS [He 15]. Nonetheless, the amount of convolutions in one network have risen as high as 154, thus maintaining the importance of optimization of the convolution at inference time.

Despite the advantages of residual neural networks, this work will show that the low-precision arithmetic technique fails for ResNets (and for not very well-established reasons). The situation calls for further investigation, as the potential of these innovative networks has shown to be immense [Simo 14].

This work is organized as follows. In Chapter 2, a background on the various processes and technologies involved in this research are outlined. The reader will be familiarized with core concepts, namely those that contribute to the workings of convolution and low precision in neural networks.

In Chapter 3, the experimental methodology is given. The processes involved for setting up the experiments and for designing the elements of the system are explained.

Chapter 4 gives a detailed account of the experimental results in regards to changes in accuracies and speeds in the context of different types of arithmetic precision. Results from auxiliary experiments on an embedded system, a Raspberry Pi 3, are also given.

Chapter 5 serves as a conclusion, and will mention both system and experimental limitations, as well as items which were found to be of most immediate importance in regards to future research.

Chapter 2

Background and past work

This chapter will familiarize the reader with the various topics, strategies and technologies involved in this research. Namely, convolutional neural networks, convolutions, computer precision and integer representation, and other relevant topics will be discussed.

2.1 Past work

Previous work addressing the neural network number-representation problem has shown quantizing values to integers, or in some cases only as far as fixed-point representation, to be effective. One such example is research born out of the Watson Research Center and Almaden Research Center [Gupt 15]. This work shows that it is possible to reduce representation to 16-bit not only at inference time, but at training time, and without incurring a loss in accuracy. This furthers the case for the robustness of convolutional neural networks and highlights the potential for quantization of networks at inference. A second example is a work in which weights are kept in floating-point representation while activations are converted to fixed-point representation. This paper makes the claim that number representation range (the ability to represent large and small values) is more important than representation precision. [Lai 17]. Google has also made great strides in regards to the quantization topic. One work emphasizes the real-time intensity of neural network processing through the example of speech recognition, and succeeds in optimizing performance of such processing at runtime on CPUs [Vanh 11b].

2.2 Image recognition

Image recognition is a subtask in the field of computer vision in which the goal is to correctly predict a text label for the most dominant object in the image [Ha T 17]. Convolutional neural networks are designed to handle the image recognition problem. They got their start at the ILSVRC challenege in 2012, where AlexNet, the first CNN, won by a significant factor [Ha T 17]. Since then, convolutional neural network architectures have been modified and experimented with, keeping the core concept of the convolution layer in mind. As a result of this research, there now exist CNNs which perform at near-human levels. One example is the popular LeNet architecture [LeCu 10], where top-5 accuracy for the task of recognizing hand-written numbers has reached 100%. Another example is the family of residual neural networks, whose very deep architectures and interwoven stacks of convolution layers have achieved an error of only 3.57% on the famous Imagenet dataset [He 15].

2.3 Convolutional neural networks

2.3.1 Motivation

2.3.1.1 The limitations of non-convolutional neural networks

In a traditional neural network, each node in each layer is fully-connected to each node in the preceding layer. This means that every activation is derived from the full scope of input from the previous layer, making the process computationally expensive and therefore slow. The memory needed for such a network is staggering. Imagine a modestly-sized network that takes images of size 224×224 and three channels (RGB), with perhaps only two hidden layers comprising of 4096 and 1000 nodes, sequentially. All together this network would require $224 \times 224 \times 3 \approx 150K$ values for the input layer, $\tilde{4}$K and 1K values to represent the first and second hidden layers, with $224 \times 224 \times 3 \times 4096 \approx 616M$ and $4096 \times 1000 \approx 4M$ values in weights for these hidden layers, respectively. This would amount to a total of $775M \times 4 \approx 3GB$ of storage needed—at floating-point precision—for this relatively simple network. It is also of note that in such a scheme in which there is a large amount of independent and individual parameters, it would be difficult not to overfit the data at training time. One can easily see that the traditional non-convolutional network is not ideal for image input. Although large CNN architectures such as VGG

can reach a similar number of parameters, they are at the same time much deeper and expressive. In the case of VGG-19, the fully-connected layer (usually occurring at the end of a CNN network) brings the total parameter count to approximately 138.3M [Chat 14]. However, a similar multi-layer perceptron network of similar layer-size would far exceed this number.

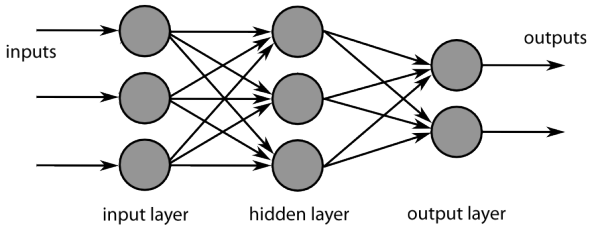

Figure 2.1: A traditional neural network. Notice how each node is connected to every previous node. Weights add up quickly and the image recognition problem soon becomes unmanageable. [CC Image courtesy of Chrislb on Wikimedia Commons]

2.3.1.2 Convolutional neural networks as a solution

Convolutional neural networks approach the image recognition problem differently. Namely, they make one main assumption and one crucial adjustment. These networks assume that the input is an image, and nothing else. They then introduce parameter recycling and the replacement of fully-connected layers with convolutions. Because of the nature of images, sharing of parameters across the network is possible, reducing their number drastically. By expecting image input, and by recognizing the regular "nature of images" to be consistent, one can build in premeditated adjustments to the network through various techniques [Karp 17]. To understand why such an assumption works, one can consider how one section of an image may share properties with another section of the image. It is possible in such a situation to replace the fully-connected "filter" with local and smaller filters (e.g. 3×3), only looking at one section of the image at a time. This filter can then be moved, *with the same weight parameters*, to other sections of the image, keeping in mind that if the filter works at one location in the image it should also be applicable at another location. This again is realizable due to the aforementioned "nature of images": namely that they share properties throughout their field [Karp 17]. Replacing our first fully-connected layer in our previous example with a reasonable convolution layer with, say, 64 outputs and a filter-size of 3×3 would result in a weight parameter matrix of only size $3 \times 3 \times 3 \times 64 \approx 1.7K$.

2.3.2 Architecture

2.3.2.1 Overview

Popular convolutional neural network models, although diverse, tend to vary in systematic ways. In their most basic form, CNNs may be comprised of only a few types of layers: the input layer, the convolution, the pooling layer, activation layers (e.g. ReLU), and the fully-connected layer [Karp 17]. Typically, most of the heavy lifting in terms of computational resources will be, by a strong margin, performed by the convolutional layers. Famous convolutional neural network architectures include AlexNet, GooglNet, ResNet and VGG, all of which differ in layer number, layer ordering, and convolution layer parameters, such as the number of filters, the size of the filters, and the stride of the filters. These layers are arranged in varying fashion to make up the skeletons of the well-known convolutional neural networks. To see some typical architectures outlined, refer to Table 2.1 on page 8.

2.3.2.2 The convolution layer

The convolution layer is the cornerstone of the convolutional neural network, and is made up of several components.

The first component of a layer l is the layer activation itself (the layer output). The general form of a layer activation a_l in a convolutional neural network can be interpreted as a three-dimensional tensor $X \times Y \times D$ where X and Y are the activation dimensions and D, or the activation depth, is equal to the number of kernels applied to the input.

The second component, the input of the current layer a_l, is equivalent to the activation of the previous layer as follows:

$$input_{a_l} = a_{l-1}. \tag{2.1}$$

Note that the initial image or first layer of a network is an $X \times Y \times 3$ tensor, where D in this case represents the RGB color channels of the original image. Beyond the input layer, the value of D at a_l is then the number of kernels just applied to the input a_{l-1}.

The third component is the kernel set, made up of D kernels. Each kernel $kernel_d$ of the kernel set is a three-dimensional tensor and is of shape $M \times M \times Z$, where M, representing

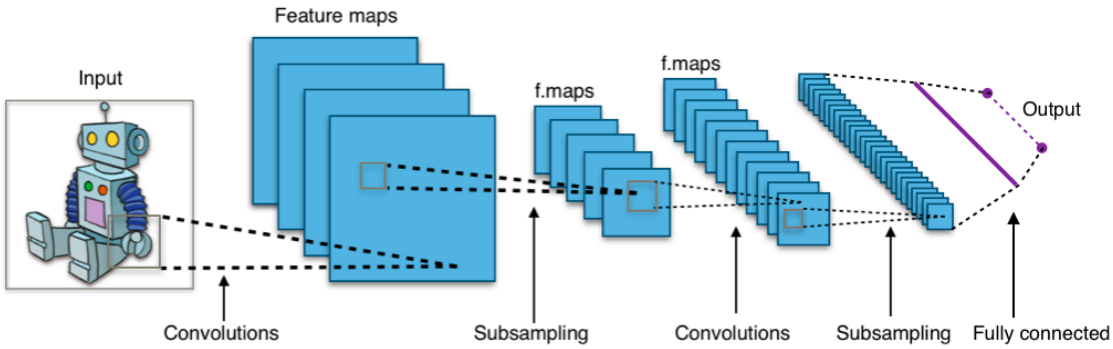

Figure 2.2: A basic convolutional neural network. Weight parameters are stored only for a predetermined amount of filters (here, "f.maps") of a certain dimension. For example, shown are five filters of size 5×5 for one convolution, followed by twelve filters of size 3×3 for the second). [CC Image courtesy of Aphex34 on Wikimedia Commons]

Table 2.1: The basic outline of some well-known CNN architectures.

(ReLU layers omitted)

LeNet	**cifar-10**	**VGG-16**	**VGG-19**
conv	conv	conv	conv
pool	pool	conv	conv
conv	conv	pool	pool
pool	pool	conv	conv
fc	conv	conv	conv
fc	pool	pool	pool
	fc	conv	conv
	fc	conv	conv
		conv	conv
		pool	conv
		conv	pool
		conv	conv
		conv	conv
		pool	conv
		conv	conv
		conv	pool
		conv	conv
		pool	conv
		fc	conv
		fc	conv
		fc	pool
			fc
			fc
			fc

the kernel width and height, is of a predetermined dimension (usually small, e.g. 3, or perhaps 7). Z is the kernel depth, and is equivalent to D_{l-1}, or the previous activation's depth. Each kernel $kernel_d$ makes up a three-dimensional tensor reaching through the entire depth Z of the input. The role of each kernel is to slide across the input at a predetermined "stride", much like a moving window, gathering a dot product at each slice of itself along Z with the current input lying within this window, and summing each of these products to return a single value [Karp 17]. Each of the kernels $kernel_d$ performs its task as described above, delivering a two-dimensional matrix product result. These two-dimensional results are then stacked along D to form the three-dimensional volume a_l seen in Figure 2.3.

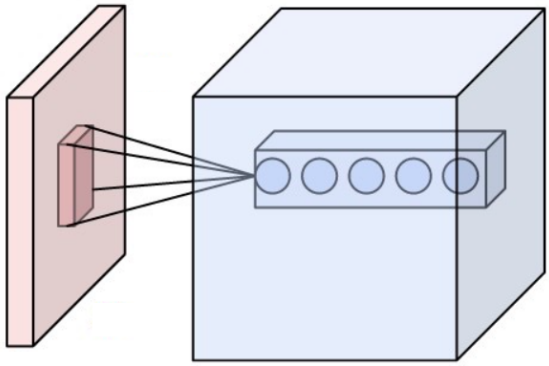

Figure 2.3: Each set of kernels (here, of count five) focuses on a small section of the input (in pink) at a time, reaching through its entire depth. The result from each of these sets of three-dimensional kernels is a stack of two-dimensional activations, making up one three-dimensional section (small rectangle in blue) of the total output volume (large box in blue), having the depth of the number of kernels in the set. This operation is repeated all over the image. [CC Image courtesy of Aphex34 on Wikimedia Commons]

The remaining components make up what can be viewed as the hyperparameters of the kernel, and include its shape, its stride and the amount—if any—of padding it adds to the input layer. A typical size is 3×3, and a typical stride is one.

During the forward pass, each element $a_l[x, y, d]$ is computed by applying the $kernel_d$ to a region of a_{l-1} of size $M \times M \times Z$, performing dot products along each dimension Z and summing these products. After each such calculation, the kernel moves along from left-to-right, top-to-bottom at an increment equal to the stride parameter. This is then repeated for all kernels and the results are stacked along D. Equation 2.2 below shows this operation for one kernel, assuming that stride is set to one and there is no zero-padding. In practice, convolution reduces the size of the input volume along the length and width dimensions, and adding padding beforehand can amend this.

$$a_l[x, y, d] = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \sum_{z=0}^{Z-1} a_{l-1}[x + i, y + j, z] * kernel_d[i, j, z] \tag{2.2}$$

2.3.2.3 Pooling and relu layers

Pooling layers are also important in understanding how CNNs change the input volume. They are different in that they don't contain weights, and are therefore lightweight and not a natural candidate for optimization like the convolution layer.

The pooling layer takes three required hyperparameters: pooling size, pooling mode and stride (for the sake of simplicity, this description will assume a padding of zero). Pooling size, or the pooling dimensions a and b, define the size of the pooling kernel, much like the kernel in the convolution layer. The pooling mode is typically average-pool or max-pool [Karp 17], denoting mathematical operations of mean-value or maximum-value, respectively.

The layer takes an input volume p of size $[x, y]$ and, assuming a stride of one, returns a reduced input volume p' of size $\frac{x}{a}, \frac{y}{b}$, where a and b are the kernel dimensions. To determine an element at the output volume p', with a pooling mode of maximum-value, a kernel with dimensions $a \times b$ and a stride of one, the pooling layer performs the following operation:

$$p'[x, y] = \max_{i=0}^{a-1} \max_{j=0}^{b-1} p[x + i, y + j]. \tag{2.3}$$

Notice that unlike convolution, the are no kernel parameters. The pooling layer serves to reduce the size of the input layer and in this sense, can indirectly have a dramatic effect on the speed of the network by reducing the input size of a subsequent convolution layer, thus reducing the total number of FLOPS needed for such an operation.

ReLU layers, like pooling layers, don't contain parameters. However unlike pooling layers, they do not change the input volume size. Their purpose is to apply an element-wise non-linearity to a layer, and are somewhat uninteresting in the context of this work.

2.4 GEMM

As previously mentioned, in the case of the famous AlexNet architecture, 89% of computational processing time is taken up by convolutions. Thus, it is a worthy undertaking to optimize the mathematical operation itself, making it as efficient as possible for the computer. The agreed upon solution is the GEMM operation, or General Matrix to Matrix Multiplication, dating back to 1979 and described in "Basic Linear Algebra Subprograms for Fortran Usage," or BLAS [Foun 02]. BLAS improves upon the normal convolution operation by transforming it into one single matrix-matrix multiplication. Normally, a convolution involves the process described in Equation 2.2 on page 10, where a filter performs tiny matrix-matrix multiplications all over the image. Instead, GEMM performs the "image-to-column" operation (commonly known as "im2col") [Chel 06], successfully translating the input volume and weight kernels such that the result of a convolution involves one and only one matrix multiplication between the im2col output of input and the im2col output of weights.

Behind the scenes, im2col is simply serializing each input selection (small pink box in Figure 2.3 on page 9) into one row of the new input matrix. Similarly, each kernel is serialized as a column of the new weight matrix. The result is now a classic matrix-matrix multiplication, visualized in Figure 2.4. Effectively, performing im2col and then matrix multiplication is equivalent to performing convolution as described in 2.2 on page 10 for all values of k.

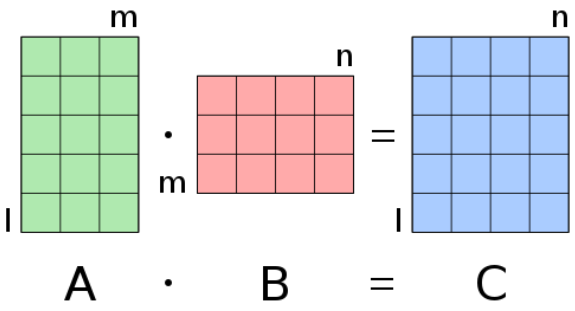

Figure 2.4: After GEMM prepares the input and weight volumes, the result of a convolution is obtained simply through traditional matrix-matrix multiplication. [CC Image courtesy of Quartl on Wikimedia Commons]

In the next section, now with a firm understanding of the inner workings of convolutional neural networks and the convolution operation itself, quantization, as a method to reduce computational overhead and increase inference speed, is discussed.

2.5 Quantization

2.5.1 Computer number representation

In order to understand why quantization is important, it would be beneficial to review how the computer stores numbers.

32-bit floating-point representation is a computer representation of real numbers. Although an estimation in itself—as real numbers are infinite and computer-representations cannot be—they offer a certain degree of precision in number representation and calculations [Elec 85]. In a computer, they take scientific notation form (see Equation 2.4 below) and can represent 2^{32} individual values. They use one bit for the sign, eight bits for the exponent, and 23 bits for the fraction [Elec 85].

$$-9.876 = \overbrace{\underbrace{-1}_{\text{sign}}}^{\text{1 bit}} \times \overbrace{\underbrace{9876}_{\text{fraction}}}^{\text{23 bits}} \times \overbrace{\underbrace{2^{-3}}_{\text{exponent}}}^{\text{8 bits}} \tag{2.4}$$

Integers, on the other hand, are represented by a fixed amount of bits, such as eight [Elec 85]. 32-bit/ 64-bit processors are able to access large chunks of memory at a time, and it would in theory be faster to use integers rather than floating-point representation, as $\times 4$ the amount of numbers would be accessed within a single memory-retrieval operation, reducing memory bandwidth by 75%. However, 8-bit integers have the potential to represent only 2^8 distinct numbers, and thus using them comes at a precision cost. It will be shown, despite this significantly reduced precision, that using integer representation in convolution calculations is still worthwhile.

2.5.2 The quantization process

In short, quantization is a conversion of floating-point representation, specifically 32-bit floating-point, to integer representation, or 8-bit fixed-point. The process of quantization is relatively straightforward. Taking the minimum and maximum of the floating-point representation, a new range is defined using an appropriate integer representation, such as 0 to 255. [Ward 16]. In other words, 0 will now represent the minimum value from the original unquantized matrix and 255 the maximum value (see Table 2.2 below).

Table 2.2: Quantized value representation.

32-bit	**8-bit**
-2.356	0
1.201	127
4.758	255

2.5.3 The quantization process broken down

Mathematically, the conversion from a floating-point tensor f to integer representation q involves a few processes, the first of which is to derive a quantization scale parameter from the minimum and maximum of the floating-point representation as follows:

$$scale = \frac{\max_f - \min_f}{255 - 0}. \tag{2.5}$$

Next, the zero-point of the quantized representation is determined and rounded as an integer,

$$zeropoint_q = round(\min_f - \frac{\min_q}{scale}), \tag{2.6}$$

and finally the floating-point values are converted to integers with the scale and zero point parameters,

$$q[x, y, z] = zeropoint_q + \frac{f[x, y, z]}{scale}, \tag{2.7}$$

and confined within the constraints of the defined quantized range if they happen to fall outside of it [Goog 17] like so:

$$\begin{aligned} u &= \min_{255, q[x,y,z]} \\ q[x, y, z]_{confined} &= \max_{0,u}, \end{aligned} \tag{2.8}$$

where x, y and z represent tensor coordinates.

After quantization, calculations are performed as usual. The result is then converted back into 32-bit float using more minimum and maximum parameters and passed along the network.

It may be apparent to the reader at this point that there is not insignificant overhead involved in this process: namely, determining parameters and converting back and forth between quantized and dequantized representations for each convolution layer. This will be addressed later in this work.

Chapter 3

Methodology

In this chapter, the experiment's methodology is summarized and abstracted in regards to the research goals, experimental setup and program implementation of the work. The process of preparing and benchmarking neural networks with varying arithmetic precisions, as well as the creation of an end-to-end system for carrying out these tests, is detailed. In brief, trained models are obtained from the scientific community for each architecture form various resources, model weights are extracted and inference calculations are made at different precisions.

3.1 Experimental summary

The goal of this work is to study how performing inference calculations with reduced arithmetic precision affects two measures: runtime and accuracy. Given a convolutional neural network model M, runtime R and accuracy A of inference with M are calculated with a varying GEMM mode $C \in eigen, gemmlowp$ where *eigen* [Guen 10] uses 32-bit floating-point representation and *gemmlowp* [Goog 17] uses quantized 8-bit integers. *gemmlowp* also performs quantization, the process of which is detailed in Section 2.5.3. A reference implementation using Caffe [Jia 14] was implemented, as described in Section 3.5.1.

Concretely, formula 2.2 on page 10 is either computed in full float precision with eigen or with quantized values with gemmlowp as described in the formulas in Section 2.5.3.

3.2 Model architectures

The models used in this work were chosen because they represent small, medium and large model sizes in terms of *the number of trainable weight parameters*. Additionally, in their number of convolutions, they trend upwards, generally exponentially, which creates a natural environment for the comparison of runtimes of convolutions (see Figure 3.1). "LeNet" [LeCu 10] and "cifar-10_quick" [Kriz 09] represent the two smaller models. The larger models include the popular "VGG-16" and the even larger "VGG-19" [Chat 14]. The medium-sized models included in experimentation consist of the revolutionary "ResNet50" and "ResNet101" [He 15] models. For the purpose of the experiments the convolution layers are of most interest. Each model has a different number of these layers. LeNet has a mere two such layers, cifar-10_quick has three, and VGG-16 and VGG-19 have 13 and 16 convolution layers, respectively. Refer to Table 2.1 on page 8 for the layer outline of these model architectures. ResNet50 and ResNet101 have—by far—the most number of layers, totaling 53 such operations for ResNet50, and 104 operations for ResNet101.

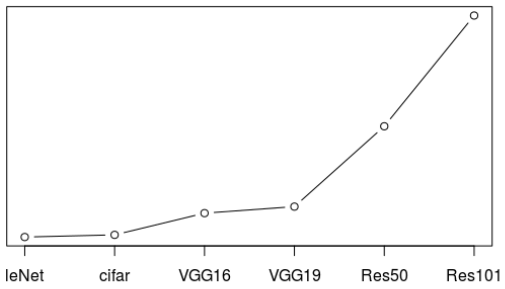

Figure 3.1: The relationship between convolution counts for the models used.

The model architectures and their relevant properties are highlighted in Table 3.1 on page 17, while some diagrams of varying model designs are displayed in Figures 3.2 and 3.3.

3.3 Inference inputs

The size and nature of inputs also differs amongst models used. The LeNet models uses its famous MNIST handwritten database dataset of hand-written digits [LeCu 10]. These are one-channel (gayscale) images of size 28×28. The cifar-10_quick model is fed

Figure 3.2: Design of the VGG-16 architecture. [Image courtesy of book.paddlepaddle.org]

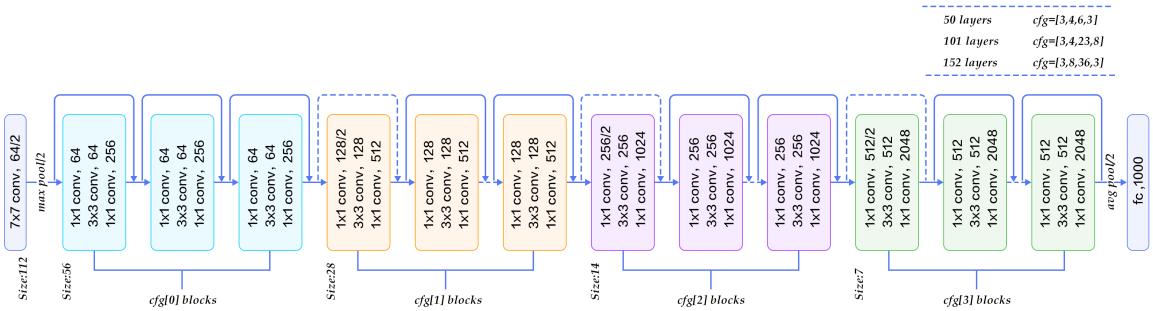

Figure 3.3: Design of the ResNet architecture. [Image courtesy of book.paddlepaddle.org]

Table 3.1: The models and their properties.

Model	layers	params	input	convs	pools	products
LeNet	4	430.5K	1x28x28	2	2	2
cifar-10	5	145.4K	3x32x32	3	3	2
VGG-16	16	138.3M	3x224x224	13	5	3
VGG-19	19	143.7M	3x224x224	16	5	3
ResNet50	50	25.6M	3x224x224	53	2	1
ResNet101	101	44.5M	3x224x224	104	4	1

with 32×32 3-channel (RGB) images. They are mean-image normalized. All four other models take ImageNet images as input, specifically the first 1000 images of the ILSVRC 2012 competition's validation set [Deng 09]. They are either mean-pixel or mean-image normalized, depending on what was done at training-time, and are of size 224×224.

3.4 Breaking down the low-precision process

The requirements for integrating low precision into convolutions can be broken down into several components. Some of these can—and are—obtained or performed offline, while others are strictly online run-time processes. These components are, briefly: result matrix quantization parameters, input quantization parameters, weight quantization parameters, the actual quantization process for inputs, activations and weights, the GEMM operation, and dequantization. Table 3.2 on page 20 shows the different steps involved, whether they can be performed offline, and whether they are involved in calculating timings for the experiments. The quantization method is highlighted in Section 2.5.3.

3.5 Implementation details

3.5.1 Caffe feature and weight extraction

Caffe is a popular deep-learning framework born out of Berkeley College [Jia 14]. It has a vast number of publicly-accessible models made available, and processes in 32-bit float [Jia 14], thus making it a good candidate for building into our custom system as a means to select models, provide model weights and baseline features, and reference against the custom implementation to make sure it is bug-free.

At this point, it may also be of use to keep track of the number representations of the various libraries involved in extracting weights and features, as to be able to effectively compare to the Caffe baseline, and to keep our weights in the right precision.

Weights are extracted from the forward-pass of the Caffe network, instantiated through Caffe's "Caffe.Net()" method, taking a .prototxt and .caffemodel file and called with "Caffe.Net.().forward()" [Jia 14]. Caffe processes in single floating-point precision, so the current state is 32-bit. After extraction, weights are saved via Python's Numpy package, which uses double precision [Jone 01], so our precision is maintained.

First in the pipeline of the Caffe feature extraction is, after the forward pass, the saving of activations by Python's Numpy [Jone 01]. Therefore the reference features are of the correct precision and are comparable to the 32-bit experiment results.

The inputs, having values typically ranging from 0-255, are inherently integers and remain so until activated in the first layer of the network (or rather are represented by floats with a trailing ".0", unless quantized), where they either remain integers (in integer mode) or become floats due to multiplication with weights.

3.5.2 Inference

All experiments were run on an Intel Core i5-2520M CPU @ 2.50GHz x 4 with 3.7 GB of RAM. The operating system was Ubuntu 16.04 LTS 64-bit. Additionally, auxiliary experiments were run on a Raspberry Pi 3 (see Section **??**.

In both modes, input is run through the custom system related to this work. The system includes bare-bones implementations of all relevant layers: convolution, pooling, ReLU, fully-connected, batch-normalization, scale, and eltwise. The inference files and makefiles are automatically generated via a script-generation script written in Python, which parses the Caffe prototxt files and outputs a C++ script in the most minimal way possible. Great care has been taken in ensuring that the script is minimal: all large objects are passed-by-reference when possible, eigen routines are called conservatively, the loading of parameters, weights and inputs, handled by the Armadillo package [Ho 12], is optimized, and the compiler is set to optimize for speed. The options used at compile time are as follows:

-c -O3 -march=native -std=c++11

The option "-O3" sets the compiler to make full optimization: it attempts to reduce code size and execution time, and performs all other optimizations possible. Furthermore, "-march=native" tells the compiler to use the platform-specific assembly code instructions.

3.5.3 Integrating low precision

The low-arithmetic precision module was taken from gemmlowp, a "small self-contained low-precision GEMM library" [Goog 17]. Gemmlowp performs several tasks needed for the experiments in this work. Apart from performing the actual GEMM procedure, gemmlowp

Table 3.2: Subprocesses of low-precision GEMM. Although low-precision arithmetic is in theory faster, computation time can quickly add up with other online operations.

Process	**possible offline?**	**incl. in timing?**
Calculate input matrix MIN/ MAX	yes	yes
Calculate weight matrix MIN/ MAX	yes	no
Calculate activation matrix MIN/ MAX	can estimate	no
Quantize input matrix	yes	yes
Quantize weight matrix	yes	no
GEMM	no	yes
Dequantize activation matrix	no	yes

quantizes inputs, dequantizes, retrieves parameters needed for these quantizations, and quantizes weights. This section will detail the processes involved in replacing regular floating-point GEMM operations with a custom call to the gemmlowp module.

One of the convenient realities that makes low-precision GEMM work is the fact that many—and sometimes all—of the parameters can be calculated offline [Ward 16]. For the weights of a cnn, this is always the case, as the weight parameters are learned at training-time. Thus, the minimum and maximum of these matrices can be stored as constants. For the inputs, it is also possible to collect parameters offline. This is straightforward for standard RGB images, known to hold values in the range [0-255]. The exact processes that accompany these calculations are detailed in the equations in Section 2.5.3.

For result/ activation parameters, this is a bit more complex. One cannot know beforehand the resulting matrices at each convolution. However, it's possible to collect estimations of these parameters using the training images. In this experiment, these parameters are gathered prior to runtime using a "hold-off" set whose distinct purpose is to provide runtime parameters. This is all done offline, as it is equivalent to collecting the same parameters at training time. This cuts down on timing and apparently doesn't cost much in terms of accuracy, as is explained in the next chapter.

Quantizing inputs and weights, like finding their parameters, can also be done offline. However here, quantization parameters and quantizations of the input layer have been treated equally with other activation layers, and thus are computed online. We assume for the purposes of these experiments that the input matrix range was not known beforehand.

GEMM itself is an online operation, as well as dequantizing activation matrices, as results are of course not known until after GEMM.

Chapter 4

Experiments

In this chapter, results of the experiments are discussed as they relate to speed and accuracy of convolutional neural networks. The experimental setup is explored, and the subprocesses of low-precision GEMM are outlined. Additionally, some results on an embedded system are given.

4.1 Experimental setup

Six different model architectures were explored in this work. Some information regarding the modules used in the experiment—namely network depth, the input size, the number of convolutions and the number of trainable parameters—are showcased in Table 3.1 on page 17. The chosen models are run through the system with the inputs described in Section 3.3. First, Caffe extracts weights and features. Then the architecture is parsed from the Caffe prototxt and the inference file is generated. The script is then compiled with both eigen and gemmlowp arithmetic options, and inference runs its course. Finally, from ground-truth labels the inference results are derived, and speeds are collected.

4.2 Accuracies

4.2.1 Evaluation criteria

The first part of the experiment involves accuracy testing, where the accuracy of image recognition model inferences, run on 32-bit float precision and then integer precision, are

compared. The criteria for an accuracy here is two-fold. Both top-1 and top-5 accuracy are used. Top-1 accuracy is classic model precision. Top-5 accuracy differs in that it rewards the system a correct prediction as long as the model predicts the appropriate class *within* its top five predictions. Thus, the predictions with the 5-highest values are considered "correct predictions." This is a common heuristic in image recognition, but the reasoning is not that well defined. One suspicion is that it is to account for the fact that there can be several labels attributable to an image simultaneously, perhaps creating a situation in which there is more than one "correct" answer. In these experiments, it's used really just to adhere to image recognition evaluation standards.

4.2.2 Accuracy results

Table 4.1 on page 23 reports the results of the accuracy tests. To evaluate, the results are broken down into three sections: small-sized models results, medium-sized model results, and large-sized model results.

The integer versions of the small sized models, LeNet and cifar-10, both performed at nearly the same or exactly the same top-1 accuracy as their floating-point counterparts. In the case of cifar-10, a 0.1% reduction in accuracy was observed; however, the floating-point recreation of the Caffe baseline was already 0.1% higher than the baseline itself. Therefore, it can be said that integer multiplication in the case of this model suffers no loss of accuracy from the Caffe baseline.

The large models, VGG-16 and VGG-19, also displayed fantastic results, the latter even gaining 0.1% accuracy from the floating-point eigen implementation. VGG-16 performance was reduced by a negligible 0.1%.

In terms of top-5 accuracy, LeNet and cifar-10 stayed steady relative to the floating-point counterparts, while VGG-16 and VGG-19 both deviated, the former suffering a 0.3% loss, and the latter gaining 0.4% in accuracy.

The last models, ResNet50 and ResNet101, did not perform well with integer arithmetic. Run with gemmlowp, the top-1 results of these networks were as good as 0 0.1%. Residual neural networks operate on decidedly different principles: namely, they refer back to old convolution results earlier in the network as participants in later convolutions [He 15]. This modification, being the most profound perceivable architectural difference setting it apart from other types of CNNs, apparently causes integer arithmetic, as it's performed by gemmlowp, to fail completely. This is perhaps because inputs to a given convolution are not necessarily taken from the previous layer's output, but often from layers prior. As these

Table 4.1: The accuracy results.

model/ mode	top-1 (%)	top-5 (%)
LeNet		
caffe	97.6	100
eigen	97.6	100
gemmlowp	97.6	100
cifar-10		
caffe	74.6	98.8
eigen	74.8	98.7
gemmlowp	74.7	98.6
VGG-16		
caffe	70.2	80.8
eigen	70.2	80.8
gemmlowp	70.1	80.5
VGG-19		
caffe	69.9	80.5
eigen	69.9	80.5
gemmlowp	70.6	80.9
ResNet50		
caffe	75.2	83.5
eigen	75.2	83.5
gemmlowp	0.1	0.3
ResNet101		
caffe	75.1	84.3
eigen	75.1	84.3
gemmlowp	0	0.3

networks can be considered equivalent to many shallower networks stacked together, where tight layer relationships are not necessarily a reality, individual quantized perturbations of low-precision calculations do not carry over from layer to layer. Additionally, the "eltwise" operation of residual neural networks, where old activations are added or multiplied with more recent activations, is performed in the dequantized 32-bit representation, and this might also have an adverse effect on the network's ability to be converted to low-precision.

4.3 Speeds

4.3.1 Evaluation criteria

The second part of the experiment is speed testing. For eigen mode, the speed heuristic is simple: the time taken for GEMM multiplication. For gemmlowp, it's more complex.

In Figure 3.2 on page 20 one can see the gemmlowp call broken down by subprocess. Amongst these sub-processes, some are "online", or considered in the timing-scheme, or are "offline", meaning they are able to be performed before inference time. For these experiments, gemmlowp GEMM time is the sum of the online processes:

$$\begin{aligned}\text{total GEMM time} = {} & \text{online parameter determination} + \\ & \text{online quantization} + \text{GEMM} + \text{dequantization}\end{aligned} \tag{4.1}$$

4.3.2 Speed results

4.3.2.1 Aggregate GEMM time

Generally, the time for GEMM gained by performing quantized calculations appears to be relative to the total number of convolution parameters over the entire network. Table 4.2 on page 25 reports the average and total GEMM times for the 1000-image experiment batch. For the non-residual networks, the benefit of quantizing increases with the total number of weight parameters associated with convolution layers, and is therefore more effective with networks like VGG-16 or VGG-19 than for LeNet or cifar-10. See Table 4.3 on page 25 for the relative speed gains.

However, the speed gained with quantized GEMM in regards to the ResNet architecture is—as in the results for accuracies—irregular. These two models appear to be effected proportionally by quantized GEMM on a different scale. While they represent the models with the most number of convolution parameters, ResNet50 still reduces speed on a lesser scale than the smaller cifar-10 model. ResNet101, whose speed-gain is again postulated to respond differently than the non-residual models, receives none-the-less the most benefit from quantized GEMM.

Table 4.2: GEMM times (ms).

	batch averages		batch totals	
Model	**eigen**	**gemmlowp**	**eigen**	**gemmlowp**
LeNet	0.3744	0.278	374.4	278
cifar-10	2.1249	1.43	2124.9	1430
VGG-16	2003.53	1210	2003530	1210000
VGG-19	2464	1470	2464000	1470000
ResNet50	546.3	393	546300	393000
ResNet101	1215.48	665	1215480	665000

Table 4.3: Change in GEMM speeds with 8-bit integer mode

Model	**parameters**	**conv parameters**	**GEMM speed gain**
LeNet	430500	25500	26.75 %
cifar-10	145376	79200	32.70 %
VGG-16	138344128	14710464	39.61 %
VGG-19	143652544	20018880	40.34 %
ResNet50	25556032	23454912	28.06 %
ResNet101	44548160	42394816	45.29 %

With two convolutions, gemmlowp decreased the time needed for LeNet GEMM by 25%. Cifar-10, with four convolutions, benefited from a 33% decrease. VGG-16 and VGG-19, being fairly similar in parameter count, with 13 and 16 convolutions, respectively, both decreased their GEMM processing times by 40%. ResNet50 with 53 convolutions reduced GEMM by 28%, while ResNet101 with 104 convolutions reduced by 45%. Table 4.3 above shows the GEMM speed gain relative to model architecture.

4.3.2.2 Inference runtime

Total inference runtimes increased with gemmlowp mode. Although GEMM speeds up, the other processes detailed in Section 2.5.3 on page 13 contribute to a total gain in runtime. Namely, as detailed in Table 4.4 on page 26, LeNet slowed down by 21%, cifar-10 by 49%, VGG-16 and VGG-19 by 8 and 9%, respectively, ResNet50 by 41%, and ResNet101 by 6%.

4.3.2.3 Improving the speed of online calculations

In order to make inference with low-precision GEMM worthwhile, the overall online process must be improved by locating a feasible candidate for optimization from the

Table 4.4: Inference runtimes (ms).

	batch averages		batch totals	
Model	**eigen**	**gemmlowp**	**eigen**	**gemmlowp**
LeNet	2.0483	2.47	2048.3	2470
cifar-10	6.8941	10.3	6894.1	10300
VGG-16	3776.17	4060	3776170	4060000
VGG-19	4261.48	4650	4261480	4650000
ResNet50	1671.49	2360	1671490	2360000
ResNet101	3122.44	3310	3122440	3310000

Table 4.5: Gemmlowp convolution timings in detail. Online timing measures are in bold.

Measure	**LeNet**	**cifar-10**	**VGG-16**	**VGG-19**	**ResNet50**	**ResNet101**
from eigen	0.1907	2.435	799.9587	800.0514	377.4687	546.6961
get params	0.0572	0.6231	577.9539	333.2199	54.7163	72.7701
qtz. offline	0.2387	0.6907	135.2755	183.7007	255.8325	409.9092
quantize	0.4235	3.1463	867.8958	979.006	223.2883	314.0266
GEMM	0.278	1.43	4060	1470	393	665
dequantize	0.0125	0.0385	25.1416	27.0573	17.1881	20.6214
to eigen	0.0284	2.435	112.6077	113.4924	64.3766	85.895

equation detailed in 4.1 on page 24.

The breakdown of the low-precision timings are shown in more detail in 4.5 on above, where online timing measures are indicated in bold. Time for quantization, as detailed in Section 2.5.3 on page 13, takes up a large proportion of total runtime, and is the obvious candidate for optimization. In the case of LeNet, 60% of the online process is taken up by quantization, 68% for cifar-10, 18% for VGG-16, 40% for VGG-19, 35% for ResNet50, and 31% for ResNet101. However, it should be noted that these breakdowns are a matter of the specific implementation of the system: namely, that of gemmlowp. Mathematically speaking, quantization should be faster than GEMM, being (for quadratic matrices of size $n \times n$) of $O(n^2)$ complexity while GEMM is of $O(n^3)$ complexity.

4.3.3 Embedded system tests

Experiments run on a Raspberry Pi 3 were decidedly slower than on the Core i5. The results of inference with gemmlowp are shown below in Tables 4.6 and 4.7.

Table 4.6: GEMM times (s) on a Raspberry Pi 3 with gemmlowp.

Model	batch averages	batch totals
LeNet	0.0139	13.9
cifar-10	0.0736	73.6
ResNet50	24.6	24600

Table 4.7: Inference runtimes (s) on a Raspberry Pi 3 with gemmlowp.

Model	batch averages	batch totals
LeNet	0.0287	28.7
cifar-10	0.121	121
ResNet50	36.400	36400

Chapter 5

Conclusion

5.1 Summary

This work shows that GEMM for convolutional neural network inference can be accelerated by as much as 45% with low-precision integer arithmetic. In addition, the experiment results demonstrate negligible loss in image recognition accuracy in the case of four popular neural network architectures. To achieve such a speed-up, quantization methods were employed. The work showcases the quantization technique for inference speed improvement in the context of the image recognition problem, whereby a lexical description is predicted given an image.

5.2 System limitations and future work

However, two architectures are concluded to be incompatible with the quantization method used. Namely, inference with residual neural networks fail with such a technique and therefore completely break down when quantized. This is speculated to be because of the nature of such networks, where inputs to a given convolution are not necessarily taken from the previous layer's output, but often from layers prior. As these networks can be considered equivalent to many shallower networks stacked together, where tight layer relationships are not necessarily a reality, individual quantized perturbations of low-precision calculations do not carry over from layer to layer.

As the size of convolutional neural networks grow, the demands for accurate classifications persist, and smaller and smaller devices like phones and other embedded systems are

expected to run image recognition software, quantization of neural networks is a research field worthy of continuation. This work demonstrates that in order to reap the benefits of low-precision GEMM, total runtime must be reduced by optimizing or off-lining the online processes involved in quantization as much as possible: online parameter calculation (i.e. maxima and minima of input and output volumes), online quantization of GEMM components, and dequantization.

Appendices

Appendix A

lpa_cnn documentation

A.1 lpa_cnn

Low Precision Arithmetic For Convolutional Neural Network Inference

`lpa_cnn` is a benchmarking tool for comparing accuracies and speeds of convolutional neural networks run with different arithmetic precision modes for the convolutions. The first mode is the baseline Caffe implentation, the second is floating point arithmetic with eigen, and the third is quantized mode, which uses integer airthmetic through gemmlowp.

A.1.0.1 Dependencies

gcc 5.4 w/ Eigen 3 & Armadillo

Python 2.7 w/ NumPy & PIL

R w/ gtools & stringr

Caffe (see `Setup` below for installation)

A.1.0.2 Setup

Install Caffe as `caffe/` (in root directory), following the guide @ https://chunml.github.io/ChunML.github.io/project/Installing-Caffe-CPU-Only/.

Have the following files in place for each desired model:

```
models/<model_name>/<model_name.caffemodel>
models/<model_name>/<model_name.prototxt>
```

adjusting the .prototxt input layer to receive one image as follows:

```
1 x <depth> x <width> x <height>
```

Have the following input file in place for each installed model:

```
inputs/<model_name>/production/<input_file_name.csv>
```

having the form:

```
<img_0_label><img_0_channel_1>...<img_0_channel_2><img_0_channel_3>
<img_1_label><img_1_channel_1>...<img_1_channel_2><img_1_channel_3>
...
```

A.1.0.3 Reproduction

To run experiments with the installed models, call `$ bash run_routine.sh`.

Results are written to `results/`.

A.1.0.4 Installing new models

A great resource for finding new Caffe models is Model Zoo @ https://github.com/BVLC/caffe/wiki/Model-Zoo

To install a new model, follow the `Setup` directions above, providing an appropriate and consistent model name as <`model_name`>.

`NOTE` that when preparing .prototxt files, `lpa_cnn` supports the following parameters:

```
layers=['convolution','pooling','relu','eltwise', 'innerproduct',
 'scale','batchnorm']
params=['num_output','pad','kernel_size','stride','bias_term','pool']
dims=['n','d','w','h','shape','input_dim']
```

`NOTE` that batch processing is not supported.

Appendix B

Detailed runtime results

Table B.1: mnist w/ eigen

measure	batch avg (ms)	batch total (ms)
(1) 1.576.25.20.25	0.0663	62.089
(2) 20.64.500.50.500	0.308	307.952
online run time	2.0483	2048.3
aggregate GEMM time	0.3744	374.4

Table B.2: mnist w/ gemmlowp

measure	batch avg (ms)	batch total (ms)
(1) 25.576.20.25.20 convert_from_eigen	0.0311	31.071
(1) 25.576.20.25.20 convert_to_eigen	0.0218	21.841
(1) 25.576.20.25.20 dequantize	0.0097	9.71
(1) 25.576.20.25.20 gemm	0.0808	80.784
(1) 25.576.20.25.20 get_params	0.0158	15.813
(1) 25.576.20.25.20 quantize	0.1274	127.362
(1) 25.576.20.25.20 quantize_offline	0.0055	5.529
(2) 500.64.50.500.50 convert_from_eigen	0.1596	159.615
(2) 500.64.50.500.50 convert_to_eigen	0.0066	6.585
(2) 500.64.50.500.50 dequantize	0.0028	2.781
(2) 500.64.50.500.50 gemm	0.1971	197.117
(2) 500.64.50.500.50 get_params	0.0414	41.415
(2) 500.64.50.500.50 quantize	0.2961	296.08
(2) 500.64.50.500.50 quantize_offline	0.2332	233.185
online run time	2.47	2470
aggregate GEMM time	0.278	278
Aggregate convert_from_eigen time	0.1907	190.686
Aggregate convert_to_eigen time	0.0284	28.426
Aggregate dequantize time	0.0125	12.491
Aggregate get_params time	0.0572	57.228
Aggregate quantize time	0.4235	423.442
Aggregate quantize_offline time	0.2387	238.714

Table B.3: cifar-10 w/ eigen

measure	batch avg (ms)	batch total (ms)
(1) 3.1024.75.32.75	0.4059	405.88
(2) 32.256.800.32.800	1.2141	1214.055
(3) 32.64.800.64.800	0.505	505.004
online run time	6.8941	6894.1
aggregate GEMM time	2.1249	2124.9

Table B.4: cifar-10 w/ gemmlowp

measure	batch avg (ms)	batch total (ms)
(1) 75.1024.32.75.32 convert_from_eigen	0.248	248.011
(1) 75.1024.32.75.32 convert_to_eigen	0.1494	149.361
(1) 75.1024.32.75.32 dequantize	0.0236	23.592
(1) 75.1024.32.75.32 gemm	0.3558	355.781
(1) 75.1024.32.75.32 get_params	0.1325	132.517
(1) 75.1024.32.75.32 quantize	0.6701	670.133
(1) 75.1024.32.75.32 quantize_offline	0.0218	21.813
(2) 800.256.32.800.32 convert_from_eigen	1.6932	1693.19
(2) 800.256.32.800.32 convert_to_eigen	0.0424	42.438
(2) 800.256.32.800.32 dequantize	0.0099	9.911
(2) 800.256.32.800.32 gemm	0.737	737.015
(2) 800.256.32.800.32 get_params	0.4244	424.374
(2) 800.256.32.800.32 quantize	1.9593	1959.26
(2) 800.256.32.800.32 quantize_offline	0.228	228.042
(3) 800.64.64.800.64 convert_from_eigen	0.4938	493.75
(3) 800.64.64.800.64 convert_to_eigen	0.0106	10.625
(3) 800.64.64.800.64 dequantize	0.005	5
(3) 800.64.64.800.64 gemm	0.3412	341.248
(3) 800.64.64.800.64 get_params	0.0662	66.187
(3) 800.64.64.800.64 quantize	0.5169	516.897
(3) 800.64.64.800.64 quantize_offline	0.4409	440.931
online run time	10.3	10300
aggregate GEMM time	1.43	1430
Aggregate convert_from_eigen time	2.435	2434.951
Aggregate convert_to_eigen time	0.2024	202.424
Aggregate dequantize time	0.0385	38.503
Aggregate get_params time	0.6231	623.078
Aggregate quantize time	3.1463	3146.29
Aggregate quantize_offline time	0.6907	690.786

Table B.5: VGG-16 w/ eigen

measure	batch avg (ms)	batch total (ms)
(1) 3.50176.27.64.27	23.2213	23221.315
(2) 64.50176.576.64.576	355.6513	355651.302
(3) 64.12544.576.128.576	127.9835	127983.497
(4) 128.12544.1152.128.1152	248.3589	248358.941
(5) 128.3136.1152.256.1152	107.5201	107520.123
(6) 256.3136.2304.256.2304	214.8025	214802.48
(7) 256.3136.2304.256.2304	216.3558	216355.804
(8) 256.784.2304.512.2304	106.1032	106103.228

(9) 512.784.4608.512.4608	209.4344	209434.414
(10) 512.784.4608.512.4608	210.1427	210142.721
(11) 512.196.4608.512.4608	58.7688	58768.826
(12) 512.196.4608.512.4608	62.281	62280.972
(13) 512.196.4608.512.4608	62.9106	62910.57
online run time	3776.17	3776170
aggregate GEMM time	2003.53	2003530

Table B.6: VGG-16 w/ gemmlowp

measure	batch avg (ms)	batch total (ms)
(1) 27.50176.64.27.64 convert_from_eigen	7.5341	7534.13
(1) 27.50176.64.27.64 get_params	1.9414	1941.4
(1) 27.50176.64.27.64 quantize_offline	0.0209	20.865
(1) 27.50176.64.27.64 quantize	13.8209	13820.93
(1) 27.50176.64.27.64 gemm	17.7136	17713.6
(1) 27.50176.64.27.64 dequantize	6.0137	6013.73
(1) 27.50176.64.27.64 convert_to_eigen	24.952	24952
(2) 576.50176.64.576.64 convert_from_eigen	292.406	292406
(2) 576.50176.64.576.64 get_params	213.8788	213878.8
(2) 576.50176.64.576.64 quantize_offline	0.3536	353.553
(2) 576.50176.64.576.64 quantize	303.188	303188
(2) 576.50176.64.576.64 gemm	192.042	192042
(2) 576.50176.64.576.64 dequantize	7.113	7113.04
(2) 576.50176.64.576.64 convert_to_eigen	29.5355	29535.5
(3) 576.12544.128.576.128 convert_from_eigen	52.7472	52747.2
(3) 576.12544.128.576.128 get_params	44.1085	44108.5
(3) 576.12544.128.576.128 quantize_offline	0.6673	667.278
(3) 576.12544.128.576.128 quantize	73.4827	73482.7
(3) 576.12544.128.576.128 gemm	74.5619	74561.9
(3) 576.12544.128.576.128 dequantize	3.7543	3754.27
(3) 576.12544.128.576.128 convert_to_eigen	13.15	13150
(4) 1152.12544.128.1152.128 convert_from_eigen	143.4083	143408.3
(4) 1152.12544.128.1152.128 get_params	154.336	154336
(4) 1152.12544.128.1152.128 quantize_offline	1.3488	1348.8
(4) 1152.12544.128.1152.128 quantize	149.506	149506
(4) 1152.12544.128.1152.128 gemm	157.71	157710
(4) 1152.12544.128.1152.128 dequantize	3.056	3055.96
(4) 1152.12544.128.1152.128 convert_to_eigen	14.7319	14731.9
(5) 1152.3136.256.1152.256 convert_from_eigen	24.2763	24276.3
(5) 1152.3136.256.1152.256 get_params	32.3949	32394.89
(5) 1152.3136.256.1152.256 quantize_offline	2.676	2675.98
(5) 1152.3136.256.1152.256 quantize	42.0715	42071.5

(5) 1152.3136.256.1152.256 gemm	68.4284	68428.4
(5) 1152.3136.256.1152.256 dequantize	1.2388	1238.8
(5) 1152.3136.256.1152.256 convert_to_eigen	4.2447	4244.73
(6) 2304.3136.256.2304.256 convert_from_eigen	61.2143	61214.3
(6) 2304.3136.256.2304.256 get_params	39.9052	39905.2
(6) 2304.3136.256.2304.256 quantize_offline	5.3888	5388.76
(6) 2304.3136.256.2304.256 quantize	80.4736	80473.6
(6) 2304.3136.256.2304.256 gemm	135.237	135237
(6) 2304.3136.256.2304.256 dequantize	1.2894	1289.35
(6) 2304.3136.256.2304.256 convert_to_eigen	6.0984	6098.4
(7) 2304.3136.256.2304.256 convert_from_eigen	62.2938	62293.8
(7) 2304.3136.256.2304.256 get_params	43.2993	43299.3
(7) 2304.3136.256.2304.256 quantize_offline	5.4318	5431.84
(7) 2304.3136.256.2304.256 quantize	80.2911	80291.1
(7) 2304.3136.256.2304.256 gemm	133.788	133788
(7) 2304.3136.256.2304.256 dequantize	1.2267	1226.66
(7) 2304.3136.256.2304.256 convert_to_eigen	8.1492	8149.16
(8) 2304.784.512.2304.512 convert_from_eigen	16.9987	16998.7
(8) 2304.784.512.2304.512 get_params	3.4937	3493.65
(8) 2304.784.512.2304.512 quantize_offline	10.792	10791.97
(8) 2304.784.512.2304.512 quantize	21.3511	21351.1
(8) 2304.784.512.2304.512 gemm	66.2195	66219.5
(8) 2304.784.512.2304.512 dequantize	0.3771	377.137
(8) 2304.784.512.2304.512 convert_to_eigen	5.8264	5826.42
(9) 4608.784.512.4608.512 convert_from_eigen	37.9402	37940.2
(9) 4608.784.512.4608.512 get_params	26.7761	26776.09
(9) 4608.784.512.4608.512 quantize_offline	21.3695	21369.5
(9) 4608.784.512.4608.512 quantize	38.4816	38481.6
(9) 4608.784.512.4608.512 gemm	130.622	130622
(9) 4608.784.512.4608.512 dequantize	0.3894	389.412
(9) 4608.784.512.4608.512 convert_to_eigen	1.9027	1902.65
(10) 4608.784.512.4608.512 convert_from_eigen	41.3845	41384.5
(10) 4608.784.512.4608.512 get_params	11.1653	11165.29
(10) 4608.784.512.4608.512 quantize_offline	21.6243	21624.3
(10) 4608.784.512.4608.512 quantize	37.865	37865
(10) 4608.784.512.4608.512 gemm	131.482	131482
(10) 4608.784.512.4608.512 dequantize	0.4001	400.07
(10) 4608.784.512.4608.512 convert_to_eigen	2.8937	2893.7
(11) 4608.196.512.4608.512 convert_from_eigen	22.9921	22992.1
(11) 4608.196.512.4608.512 get_params	2.2006	2200.6
(11) 4608.196.512.4608.512 quantize_offline	21.849	21849
(11) 4608.196.512.4608.512 quantize	9.3671	9367.05
(11) 4608.196.512.4608.512 gemm	34.7324	34732.4

(11) 4608.196.512.4608.512 dequantize	0.0895	89.51
(11) 4608.196.512.4608.512 convert_to_eigen	0.3741	374.13
(12) 4608.196.512.4608.512 convert_from_eigen	18.2158	18215.8
(12) 4608.196.512.4608.512 get_params	2.1987	2198.67
(12) 4608.196.512.4608.512 quantize_offline	21.8163	21816.3
(12) 4608.196.512.4608.512 quantize	9.1033	9103.27
(12) 4608.196.512.4608.512 gemm	34.7381	34738.1
(12) 4608.196.512.4608.512 dequantize	0.0957	95.7
(12) 4608.196.512.4608.512 convert_to_eigen	0.3763	376.261
(13) 4608.196.512.4608.512 convert_from_eigen	18.5474	18547.4
(13) 4608.196.512.4608.512 get_params	2.2554	2255.4
(13) 4608.196.512.4608.512 quantize_offline	21.9372	21937.2
(13) 4608.196.512.4608.512 quantize	8.8939	8893.9
(13) 4608.196.512.4608.512 gemm	34.8605	34860.5
(13) 4608.196.512.4608.512 dequantize	0.0979	97.868
(13) 4608.196.512.4608.512 convert_to_eigen	0.3728	372.848
online run time	4060	4060000
aggregate GEMM time	1210	1210000
Aggregate convert_from_eigen time	799.9587	799958.73
Aggregate convert_to_eigen time	112.6077	112607.699
Aggregate dequantize time	25.1416	25141.507
Aggregate get_params time	577.9539	577953.79
Aggregate quantize time	867.8958	867895.75
Aggregate quantize_offline time	135.2755	135275.346

Table B.7: VGG-19 w/ eigen

measure	batch avg (ms)	batch total (ms)
(1) 3.50176.27.64.27	23.4011	23401.131
(2) 64.50176.576.64.576	343.8158	343815.759
(3) 64.12544.576.128.576	125.4209	125420.925
(4) 128.12544.1152.128.1152	241.0483	241048.26
(5) 128.3136.1152.256.1152	104.6489	104648.88
(6) 256.3136.2304.256.2304	217.3934	217393.437
(7) 256.3136.2304.256.2304	216.2497	216249.721
(8) 256.3136.2304.256.2304	217.0939	217093.887
(9) 256.784.2304.512.2304	103.2173	103217.344
(10) 512.784.4608.512.4608	209.1753	209175.348
(11) 512.784.4608.512.4608	208.9276	208927.567
(12) 512.784.4608.512.4608	209.855	209854.96
(13) 512.196.4608.512.4608	58.7104	58710.367
(14) 512.196.4608.512.4608	61.8953	61895.349
(15) 512.196.4608.512.4608	61.3986	61398.65

(16) 512.196.4608.512.4608	61.753	61752.986
online run time	4261.48	4261480
aggregate GEMM time	2464	2464000

Table B.8: VGG-19 w/ gemmlowp

measure	batch avg (ms)	batch total (ms)
(1) 27.50176.64.27.64 convert_from_eigen	7.4713	7471.27
(1) 27.50176.64.27.64 get_params	1.9228	1922.76
(1) 27.50176.64.27.64 quantize_offline	0.0208	20.801
(1) 27.50176.64.27.64 quantize	13.8086	13808.6
(1) 27.50176.64.27.64 gemm	17.8321	17832.1
(1) 27.50176.64.27.64 dequantize	5.9911	5991.14
(1) 27.50176.64.27.64 convert_to_eigen	25.7039	25703.9
(2) 576.50176.64.576.64 convert_from_eigen	264.876	264876
(2) 576.50176.64.576.64 get_params	129.8372	129837.2
(2) 576.50176.64.576.64 quantize_offline	0.353	352.976
(2) 576.50176.64.576.64 quantize	294.324	294324
(2) 576.50176.64.576.64 gemm	179.539	179539
(2) 576.50176.64.576.64 dequantize	6.8658	6865.75
(2) 576.50176.64.576.64 convert_to_eigen	29.4322	29432.2
(3) 576.12544.128.576.128 convert_from_eigen	48.6697	48669.7
(3) 576.12544.128.576.128 get_params	33.0163	33016.3
(3) 576.12544.128.576.128 quantize_offline	0.6628	662.772
(3) 576.12544.128.576.128 quantize	73.5885	73588.5
(3) 576.12544.128.576.128 gemm	73.3934	73393.4
(3) 576.12544.128.576.128 dequantize	3.7179	3717.86
(3) 576.12544.128.576.128 convert_to_eigen	13.3661	13366.1
(4) 1152.12544.128.1152.128 convert_from_eigen	108.3697	108369.7
(4) 1152.12544.128.1152.128 get_params	42.3338	42333.8
(4) 1152.12544.128.1152.128 quantize_offline	1.3068	1306.79
(4) 1152.12544.128.1152.128 quantize	147.44	147440
(4) 1152.12544.128.1152.128 gemm	150.484	150484
(4) 1152.12544.128.1152.128 dequantize	3.0698	3069.85
(4) 1152.12544.128.1152.128 convert_to_eigen	13.4149	13414.9
(5) 1152.3136.256.1152.256 convert_from_eigen	23.7793	23779.3
(5) 1152.3136.256.1152.256 get_params	9.4803	9480.27
(5) 1152.3136.256.1152.256 quantize_offline	2.6375	2637.51
(5) 1152.3136.256.1152.256 quantize	42.3427	42342.7
(5) 1152.3136.256.1152.256 gemm	65.7379	65737.9
(5) 1152.3136.256.1152.256 dequantize	1.293	1293.03
(5) 1152.3136.256.1152.256 convert_to_eigen	4.1752	4175.21
(6) 2304.3136.256.2304.256 convert_from_eigen	51.7341	51734.1

(6) 2304.3136.256.2304.256 get_params	19.6867	19686.7
(6) 2304.3136.256.2304.256 quantize_offline	5.4012	5401.16
(6) 2304.3136.256.2304.256 quantize	79.5124	79512.4
(6) 2304.3136.256.2304.256 gemm	131.541	131541
(6) 2304.3136.256.2304.256 dequantize	1.3939	1393.89
(6) 2304.3136.256.2304.256 convert_to_eigen	5.8042	5804.2
(7) 2304.3136.256.2304.256 convert_from_eigen	51.6142	51614.2
(7) 2304.3136.256.2304.256 get_params	20.0109	20010.9
(7) 2304.3136.256.2304.256 quantize_offline	5.4243	5424.35
(7) 2304.3136.256.2304.256 quantize	79.3014	79301.4
(7) 2304.3136.256.2304.256 gemm	131.712	131712
(7) 2304.3136.256.2304.256 dequantize	1.2536	1253.56
(7) 2304.3136.256.2304.256 convert_to_eigen	5.837	5837.03
(8) 2304.3136.256.2304.256 convert_from_eigen	55.0002	55000.2
(8) 2304.3136.256.2304.256 get_params	18.8754	18875.4
(8) 2304.3136.256.2304.256 quantize_offline	5.3733	5373.34
(8) 2304.3136.256.2304.256 quantize	76.1507	76150.7
(8) 2304.3136.256.2304.256 gemm	134.835	134835
(8) 2304.3136.256.2304.256 dequantize	1.6123	1612.31
(8) 2304.3136.256.2304.256 convert_to_eigen	5.9625	5962.49
(9) 2304.784.512.2304.512 convert_from_eigen	16.2907	16290.7
(9) 2304.784.512.2304.512 get_params	7.5403	7540.28
(9) 2304.784.512.2304.512 quantize_offline	10.7541	10754.14
(9) 2304.784.512.2304.512 quantize	20.0847	20084.7
(9) 2304.784.512.2304.512 gemm	64.7505	64750.5
(9) 2304.784.512.2304.512 dequantize	0.3728	372.788
(9) 2304.784.512.2304.512 convert_to_eigen	2.1081	2108.12
(10) 4608.784.512.4608.512 convert_from_eigen	33.4505	33450.5
(10) 4608.784.512.4608.512 get_params	9.3896	9389.56
(10) 4608.784.512.4608.512 quantize_offline	21.7143	21714.3
(10) 4608.784.512.4608.512 quantize	39.0568	39056.8
(10) 4608.784.512.4608.512 gemm	128.434	128434
(10) 4608.784.512.4608.512 dequantize	0.3858	385.752
(10) 4608.784.512.4608.512 convert_to_eigen	1.9934	1993.37
(11) 4608.784.512.4608.512 convert_from_eigen	32.8695	32869.5
(11) 4608.784.512.4608.512 get_params	17.0201	17020.1
(11) 4608.784.512.4608.512 quantize_offline	21.568	21568
(11) 4608.784.512.4608.512 quantize	38.47	38470
(11) 4608.784.512.4608.512 gemm	128.34	128340
(11) 4608.784.512.4608.512 dequantize	0.3783	378.273
(11) 4608.784.512.4608.512 convert_to_eigen	2.2019	2201.88
(12) 4608.784.512.4608.512 convert_from_eigen	33.5848	33584.8
(12) 4608.784.512.4608.512 get_params	15.791	15791

(12) 4608.784.512.4608.512 quantize_offline	21.5183	21518.3
(12) 4608.784.512.4608.512 quantize	38.3057	38305.7
(12) 4608.784.512.4608.512 gemm	128.267	128267
(12) 4608.784.512.4608.512 dequantize	0.4009	400.936
(12) 4608.784.512.4608.512 convert_to_eigen	2.0135	2013.46
(13) 4608.196.512.4608.512 convert_from_eigen	17.8692	17869.2
(13) 4608.196.512.4608.512 get_params	2.1099	2109.88
(13) 4608.196.512.4608.512 quantize_offline	21.725	21725
(13) 4608.196.512.4608.512 quantize	9.6039	9603.94
(13) 4608.196.512.4608.512 gemm	34.1958	34195.8
(13) 4608.196.512.4608.512 dequantize	0.079	78.968
(13) 4608.196.512.4608.512 convert_to_eigen	0.3673	367.26
(14) 4608.196.512.4608.512 convert_from_eigen	18.1634	18163.4
(14) 4608.196.512.4608.512 get_params	2.1233	2123.3
(14) 4608.196.512.4608.512 quantize_offline	21.8579	21857.9
(14) 4608.196.512.4608.512 quantize	9.2963	9296.3
(14) 4608.196.512.4608.512 gemm	34.1777	34177.7
(14) 4608.196.512.4608.512 dequantize	0.0798	79.833
(14) 4608.196.512.4608.512 convert_to_eigen	0.3704	370.448
(15) 4608.196.512.4608.512 convert_from_eigen	18.1136	18113.6
(15) 4608.196.512.4608.512 get_params	2.0874	2087.41
(15) 4608.196.512.4608.512 quantize_offline	21.6857	21685.7
(15) 4608.196.512.4608.512 quantize	8.9595	8959.47
(15) 4608.196.512.4608.512 gemm	34.111	34111
(15) 4608.196.512.4608.512 dequantize	0.0813	81.315
(15) 4608.196.512.4608.512 convert_to_eigen	0.3749	374.859
(16) 4608.196.512.4608.512 convert_from_eigen	18.1952	18195.2
(16) 4608.196.512.4608.512 get_params	1.9949	1994.94
(16) 4608.196.512.4608.512 quantize_offline	21.6977	21697.7
(16) 4608.196.512.4608.512 quantize	8.7608	8760.82
(16) 4608.196.512.4608.512 gemm	34.1484	34148.4
(16) 4608.196.512.4608.512 dequantize	0.082	81.957
(16) 4608.196.512.4608.512 convert_to_eigen	0.3669	366.9
online run time	4650	4650000
aggregate GEMM time	1470	1470000
Aggregate convert_from_eigen time	800.0514	800051.37
Aggregate convert_to_eigen time	113.4924	113492.327
Aggregate dequantize time	27.0573	27057.212
Aggregate get_params time	333.2199	333219.8
Aggregate quantize time	979.006	979006.03
Aggregate quantize_offline time	183.7007	183700.739

Bibliography

[Auer 08] P. Auer, H. Burgsteiner, and W. Maass. "A learning rule for very simple universal approximators consisting of a single layer of perceptrons". 2008.

[Chat 14] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. "Return of the Devil in the Details: Delving Deep into Convolutional Nets". In: *British Machine Vision Conference*, 2014.

[Chel 06] K. Chellapilla, S. Puri, and P. Simard. "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[Chin 16] S. Chintala. "convnet-benchmarks". 2016.

[Deng 09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*, 2009.

[Elec 85] I. of Electrical and E. Engineers. "IEEE standard for binary floating-point arithmetic". 1985.

[Foun 02] T. N. S. Foundation. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". *ACM Trans. Math. Softw.*, 2002.

[Goog 17] A. L. S. L. I. Google Inc., Intel Corporation. "gemmlowp: a small self-contained low-precision GEMM library". 2017.

[Guen 10] G. Guennebaud, B. Jacob, *et al.* "Eigen v3". http://eigen.tuxfamily.org, 2010.

[Gupt 15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, 2015.

[Ha T 17] D. Ha The Hien. "The Modern History of Object Recognition". https://medium.com/@nikasa1889/, 2017.

[He 15] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". *arXiv preprint arXiv:1512.03385*, 2015.

[Ho 12] N. Ho. "OpenCV vs. Armadillo vs. Eigen vs. more! Round 3: pseudoinverse test". http://nghiaho.com/?p=1726, 2012.

[Jia 14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". *arXiv preprint arXiv:1408.5093*, 2014.

[Jone 01] E. Jones, T. Oliphant, P. Peterson, *et al.* "SciPy: Open source scientific tools for Python". http://www.scipy.org/, 2001.

[Karp 17] A. Karpathy. "CS231n Convolutional Neural Networks for Visual Recognition". 2017.

[Kriz 09] A. Krizhevsky. "Learning multiple layers of features from tiny images". Tech. Rep., University of Toronto, 2009.

[Lai 17] L. Lai, N. Suda, and V. Chandra. "Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations". *CoRR*, Vol. abs/1703.03073, 2017.

[LeCu 10] Y. LeCun and C. Cortes. "MNIST handwritten digit database". http://yann.lecun.com/exdb/mnist/, 2010.

[LeCu 98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, November 1998.

[Simo 14] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". *CoRR*, 2014.

[Vanh 11a] V. Vanhoucke, A. Senior, and M. Z. Mao. "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[Vanh 11b] V. Vanhoucke, A. Senior, and M. Z. Mao. "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[Ward 15] P. Warden. "Why GEMM is at the heart of deep learning". https://petewarden.com/2015/04/20/, 2015.

[Ward 16] P. Warden. "How to Quantize Neural Networks with TensorFlow". https://www.tensorflow.org/performance/quantization, 2016.