Fakultät für Angewandte Informatik
Universität Augsburg

Master Thesis

Implementation of Frequency Domain Convolution for the Caffe-Framework

Philipp Harzig

Reviewer	Prof. Dr. Rainer Lienhart
Second Reviewer	Prof. Dr. Elisabeth André
Supervisor	Christian Eggert
Date	February 2, 2016

Multimedia Computing and Computer Vision Lab
Prof. Dr. Rainer Lienhart
Fakultät für Angewandte Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.informatik.uni-augsburg.de

Abstract

Deep Convolutional Neural Networks have received a lot of attention over the past few years as a promising technique for object classification in images. In this thesis, we implemented the frequency domain convolution for the popular Caffe framework. Deep Convolutional Neural Networks suffer from long training times even on contemporary hardware, which we want to address in this thesis. Particularly, the operation performed in a convolutional layer is computationally very expensive. We replaced the traditional convolution operation by a frequency domain convolution and achieved speed-ups of 2.2× and 1.8× in the forward pass and training of a well-known award-winning Deep Convolutional Neural Network, respectively. On self-designed convolutional layers, we achieved even higher speed-ups. Thus, we reduced long training and evaluation times of Deep Convolutional Neural Networks by a considerable factor. Moreover, we obtained a constant computation time for bigger kernels in a convolution layer, therefore, facilitating new structures for Deep Convolutional Neural Networks.

Kurzbeschreibung

Tiefe Neuronale Faltungsnetze sind seit einigen Jahren im Fokus der Forschung, da sie sich besonders gut für die Klassifizierung von Objektklassen auf Bildern eignen. In dieser Arbeit haben wir die Möglichkeit erforscht die Faltung im Frequenzraum auf Tiefe Neuronale Faltungsnetze zu übertragen. Die Faltung im Frequenzbereich behebt das Problem der langen Berechnungszeiten der herkömmlichen Faltung. Wir stellen eine neue Faltungsschicht für das bekannte Caffe Framework vor, welche die Eigenschaften der Faltung im Frequenzbereich ausnutzt. Mithilfe dieser Faltungsschicht haben wir es geschafft ein bekanntes und preisgekröntes Tiefes Neuronales Netz um den Faktor 2,2 zu beschleunigen. Außerdem waren wir in der Lage den Zeitaufwand der Trainingsprozedur um das 1,8-fache zu reduzieren. Wir ermöglichen mit der Faltung im Frequenzbereich außerdem eine konstante Berechnungszeit für größere Filter. Dadurch haben wir die Voraussetzung geschafft, um das zeitintensive Entwickeln neuer Netzstrukturen mit großen Filtergrößen um einen merklichen Faktor verbessern.

Contents

1. Introduction

Artificial Neural Networks are a very powerful methodology used in machine learning to predict answers on given input data. Their structure is inspired by a biological system – the human brain, which consists of millions of neurons. A human is able to classify known objects in a matter of milliseconds while computers struggled at this – superficially simple – task for a long time. Over the past few years, Deep Convolutional Neural Networks have emerged as one of the most promising techniques for dealing with large scale learning tasks, such as classifying objects in images. While these networks were considered way to complex and hard to understand, Krizhevsky et al. showed in 2012 [16] that training a Deep Convolutional Neural Network with millions of parameters is achievable by using high-end consumer grade Graphic Processing Units (GPU). Other techniques used for the task of classifying 1000 classes of objects performed rather weak until then. The Deep Convolutional Neural Network approach won the famous ILSVRC contest with a top-5 error rate of only 15.32 % and a difference of over 10 % to the second-ranked competitor.
Since then Deep Convolutional Neural Networks have regained a lot of interest in the research community. Even though modern hardware systems become faster and faster, training and evaluating Deep Convolutional Neural Networks take a huge amount of processing power since millions of parameters have to be adjusted hundred thousands of times. Especially, the mathematical operation of a convolution is very time-consuming, leading to training times of a couple of days for the Deep Convolutional Neural Network proposed by Krizhevsky et al. in 2012. In this thesis, we set the focus on replacing the computationally demanding convolution operation by an alternative method performing the calculations in the frequency domain rather than the time domain yielding identical results. A special focus is placed on implementing this method in a way to improve computation times of Deep Convolutional Neural Networks. For this purpose, we use the popular Caffe framework [12], which is widely used for designing Deep Convolutional Neural Networks. We supply an extended version of the Caffe framework alongside with this thesis containing the implementation of the alternative convolution method.

1.1. Related Work

In this thesis, we will focus on understanding and extending the widely-used Caffe framework [12]. It is often used for designing and evaluating new Convolutional Neural

Network structures as well as for reconstructing popular Neural Networks. We will examine basic implementation fundamentals used by the developers and then use them to properly implement our expansion to the framework. Thereto, we explore implementation details of the Caffe framework convolution as reviewed in [3] by Chellapilla et al. We use the CaffeNet, which is based on the network proposed by Krizhevsky et al. [16] in 2012, to test and develop a different approach for computing convolutions. Combined with the Caffe framework, we can implement the technique of frequency domain convolution in Deep Convolutional Neural Networks as Mathieu et al. [19] and Vasilache et al. [25] already explored in 2014.

1.2. Outline

We start this thesis with establishing a basis for understanding a convolution layer within a Convolutional Neural Network in chapter 2. We explore the basics of Deep Convolutional Neural Networks, the convolution operation, a typical network structure and discuss one of the most used techniques in digital signal processing – the Fast Fourier Transform. After that, we present a method to evaluate the convolution in the frequency domain, which often performs faster than the convolution in the time domain. We then estimate possible speed-ups of convolution in the frequency domain in contrast to the convolution in the time domain used in the Caffe framework.
In chapter 3 we describe the implementation of the convolution within the Caffe framework. We first present the details of Caffe's implementation on which we establish the convolution in the frequency domain methodology. Thereto, we describe Caffe's data structure and how we preserve compatibility with Caffe. Afterward, we discuss additional data structures, which we need to prepare in order to perform the convolution in the frequency domain. On this basis, we introduce two methods for performing pointwise complex multiplications necessary for the convolution in the frequency domain. At the end of this chapter, we discuss further optimizations to improve performance of our implementation and examine adaptations we used to implement the convolution in the frequency domain on a Graphical Processing Unit.
We evaluate our approach in chapter 4. We compare the Caffe convolution with the convolution in the frequency domain. We conduct several experiments to measure the performance of the convolution in the frequency domain. We achieve speed-ups of $2\times$ on a well-known Deep Convolutional Neural Network by using the convolution in the frequency domain. After that, we train the network with both methodologies and notice similar speed-ups when using the convolution in the frequency domain.
Finally, we conclude our work in chapter 5 and discuss on how to enhance this approach in the future.

2. The Convolution Operation

In this chapter, we introduce some fundamentals needed to understand the convolution operation as performed in a Convolutional Neural Network (CNN). We begin with defining the convolution, followed by a quick overview of the architecture of a CNN and the convolution operation in Caffe. We then give an introduction into Fourier transforms needed to express a convolution in the frequency domain. Afterward, we estimate theoretical speed-ups when using the frequency domain convolution in a convolution layer. Finally, we examine if it is possible to train a CNN by using the frequency domain convolution.

2.1. Deep Convolutional Neural Networks

A Convolutional Neural Network is a certain type of feed-forward artificial neural network. Neural Networks typically consist of an input layer, hidden fully connected layers and an output layer, which we visualize in figure 2.1. We see neurons in each layer, whereas hidden and output neurons calculate a weighted sum of weights and their inputs, which are symbolized by the black lines. We see that each of these neurons is connected to *all* neurons in the preceding layer and for that reason, the layers are called fully connected layers. Instead of only having fully connected layers, CNNs also have convolution layers. An output of such a layer is not dependent on all outputs of the preceding layer. Instead, a neuron in a convolution layer only processes portions of the inputs of its layer. We call this local portion of a neuron the receptive field of a neuron. Same weights are used multiple times at different locations of the input, which is referred to as weight sharing in a CNN. The weighted sum out of local connections can be expressed by a mathematical operation called convolution. A convolution layer convolves inputs (mostly images) with a set of weights. Deep Convolutional Neural Networks (DCNN) contain more hidden layers than standard CNNs. In the following, we explain the convolution operation to understand the mechanics of an implemented convolution layer as we later describe in chapter 3.

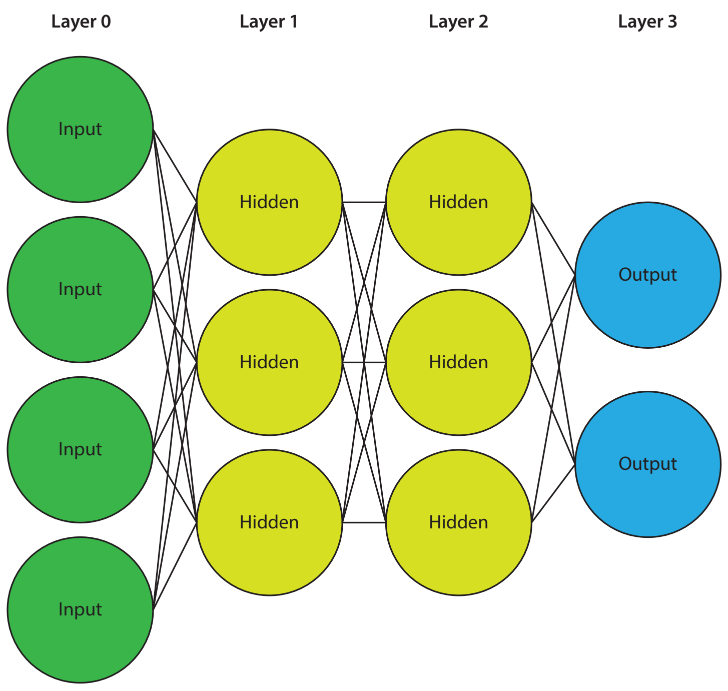


Figure 2.1.: A simple fully connected neural network with 4 inputs, 2 hidden layers with 3 neurons each and 2 outputs.

2.2. Convolution

Convolutional Neural Networks contain convolution layers that perform the convolution operation. The convolution, as described in [23], combines two signals into a third one: An input signal is combined with an impulse response yielding an output signal. The impulse response is also called a **filter**. In our case we deal with discrete signals instead of continuous signals. A discrete signal can only take a certain number of values, while there are an infinite number of other values between any two values in a continuous signal. Discrete signals in our examples also only consist of discrete values. For discrete signals the filter is applied at each point of the input signal.
Assume we have an input signal $\mathbf{i} = \begin{bmatrix} 3 & 1 & 2 & 7 & 0 & 5 & 8 & 4 \end{bmatrix}$ and a filter $\mathbf{f} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$. Before we start applying the filter to the input signal, we have to flip the filter ($\mathbf{f}' = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$) according to the definition of the convolution operation.
The first position of $\mathbf{i}$ at which we can apply the filter is 1 (we use 0-based indexing, so the value at position 1 is the second value). The center of the filter is applied at each position of the input signal and therefore cannot be applied at position 0 without zero-padding[1] the input signal. Filters commonly have an odd size of $2 \cdot n + 1, n \in \mathbb{N}$ so the filter's center is distinct. If we apply the filter at position 1, the filter spans from position 0 through 2 in the input signal. The output signal $\mathbf{o}$ is given by:

[1]Zero-padding adds zeros at the start and end of a signal to increase its length.

$$\begin{aligned}
\mathbf{o}_0 &= 3 \cdot -1 + 1 \cdot 0 + 2 \cdot 1 = -1 \\
\mathbf{o}_1 &= 1 \cdot -1 + 2 \cdot 0 + 7 \cdot 1 = 6 \\
\mathbf{o}_2 &= 2 \cdot -1 + 7 \cdot 0 + 0 \cdot 1 = -2 \\
\mathbf{o}_3 &= 7 \cdot -1 + 0 \cdot 0 + 5 \cdot 1 = -2 \\
\mathbf{o}_4 &= 0 \cdot -1 + 5 \cdot 0 + 8 \cdot 1 = 8 \\
\mathbf{o}_5 &= 5 \cdot -1 + 8 \cdot 0 + 4 \cdot 1 = -1
\end{aligned}$$

The valid[2] convolution (expressed by $*$) of the input signal $\mathbf{i}$ with the filter $\mathbf{f}$ yields:

$$\mathbf{o} = \mathbf{i} * \mathbf{f} = \begin{bmatrix} -1 & 6 & -2 & -2 & 8 & -1 \end{bmatrix}$$

The valid convolution for discrete one-dimensional signals can be defined as follows:

$$(\mathbf{i} * \mathbf{f})_{x'} = \sum_{n=0}^{N-1} \mathbf{i}_{x+(n-\frac{N-1}{2})} \cdot \mathbf{f}_{N-(n+1)} \tag{2.1}$$

N is the filter size. Our example filter $\mathbf{f}$ has a size of $N = 3$. x' is the destination index and defined as $x' = x - \frac{N-1}{2}$. If we do not zero-pad an input signal, the convolution can not be applied at all positions of the input signal since the filter is always applied at its center. Following MATLAB terminology we call this a valid convolution. The first valid position is $x_{first} = \frac{N-1}{2}$ and the last valid position is $x_{last} = \dim(\mathbf{i}) - (\frac{N-1}{2} + 1)$. $\dim(\mathbf{i})$ denotes the number of elements within the vector $\mathbf{i}$. In our example from above we applied the filter at six positions ranging from $x_{first} = \frac{3-1}{2} = 1$ to $x_{last} = 8 - [\frac{3-1}{2} + 1] = 6$. The output size is also determined by the sizes of the input signal and the filter: $\dim(\mathbf{o}) = x_{last} - x_{first} + 1 = \dim(\mathbf{i}) - N + 1 = \dim(\mathbf{i}) - \dim(\mathbf{f}) + 1$.

Convolutional Neural Networks take images as input. To convolve images, we have to define the two-dimensional convolution. A filter of a 2D convolution is often also referred to as a **kernel**. This kernel is a matrix with a size of $K_{height} \times K_{width}$. An example for a 2D kernel is given below:

$$\mathbf{K} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

This specific $\mathbf{K}$ is also known as the x-component of the Sobel Operator. The sizes of $\mathbf{K}$ in our example are $K_h = K_{height} = 3$ and $K_w = K_{width} = 3$. The two-dimensional convolution is similar to the one-dimensional convolution. We apply the kernel at every

[2]A valid convolution returns only the parts of a convolution that can be evaluated without zero-padded borders.

possible position within the given input signal **I**.

$$\mathbf{I} = \begin{bmatrix} 3 & 2 & 1 & 9 \\ 1 & 0 & 2 & 1 \\ 0 & 1 & 7 & 5 \\ 3 & 2 & 1 & 3 \end{bmatrix}$$

The only difference is, that we now have two dimensions. The possible positions range from $x_{first} = 1$ to $x_{last} = 2$ for the x-Dimension and from $y_{first} = 1$ to $y_{last} = 2$ for the y-Dimension. The output signal has a size of 2×2:

$$\begin{aligned}
\mathbf{O}_{0,0} &= 3 \cdot 1 + 2 \cdot 0 + 1 \cdot -1 + 1 \cdot 2 + 0 \cdot 0 + 2 \cdot -2 + 0 \cdot 1 + 1 \cdot 0 + 7 \cdot -1 = -7 \\
\mathbf{O}_{1,0} &= 2 \cdot 1 + 1 \cdot 0 + 9 \cdot -1 + 0 \cdot 2 + 2 \cdot 0 + 1 \cdot -2 + 1 \cdot 1 + 7 \cdot 0 + 5 \cdot -1 = -13 \\
\mathbf{O}_{1,0} &= 1 \cdot 1 + 0 \cdot 0 + 2 \cdot -1 + 0 \cdot 2 + 1 \cdot 0 + 7 \cdot -2 + 3 \cdot 1 + 2 \cdot 0 + 1 \cdot -1 = -13 \\
\mathbf{O}_{1,1} &= 0 \cdot 1 + 2 \cdot 0 + 1 \cdot -1 + 1 \cdot 2 + 7 \cdot 0 + 5 \cdot -2 + 2 \cdot 1 + 1 \cdot 0 + 3 \cdot -1 = -10 \\
\Rightarrow \mathbf{O} &= \begin{bmatrix} -7 & -13 \\ -13 & -10 \end{bmatrix}
\end{aligned}$$

We can define the 2D convolution by expanding (2.1) to the second dimension:

$$(\mathbf{I} * \mathbf{K})_{y',x'} = \sum_{h=0}^{K_h-1} \sum_{w=0}^{K_w-1} \mathbf{I}_{y+(h-\frac{K_h-1}{2}),x+(w-\frac{K_w-1}{2})} \cdot \mathbf{K}_{K_h-(h+1),K_w-(w+1)} \tag{2.2}$$

We can now apply kernels to regular images like in Figure 2.2. Convolutional Neural Networks perform many of those convolutions in every of their convolution layers.

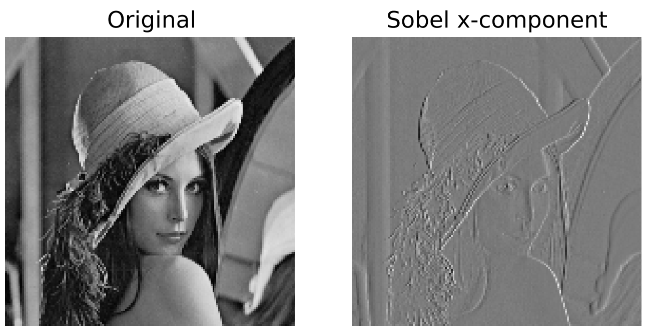


Figure 2.2.: On the left is the input image. On the right is the convolution result of the input with the x-component of the Sobel operator.

2.3. Architecture in Caffe

Throughout the thesis we use the reference network of the Caffe framework [12] as our guideline network. The reference Caffe network is based on the winner of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of the year 2012 [22]. The winner was the famous DCNN of Krizhevsky et al. [16] which outperformed every other competitor in the object detection challenge. This network is often also simply referred to as AlexNet.

We see the network design of the AlexNet in figure 2.3. The network consists of 5 convolutional layers (there are 8 learned layers in total – 5 convolutional layers and 3 fully connected layers). The first convolutional layer takes images of size $224 \times 224 \times 3$ as input. 96 different kernels of size $11 \times 11 \times 3$ together with a stride of 4 yield an output of $96 \times 55 \times 55$. This output is processed by the second layer with 256 kernels of size $5 \times 5 \times 48$. We see that the depth dimensions of the input data and the kernels do not agree. This is due to a design choice made by Krizhevsky et al., because a big network did not fit on a single GPU for training purposes. The trick was to split up the network to fit on two GPUs. He simply convolved the first part of the input data ($48 \times 55 \times 55$) with the first half of the kernels (128 kernels of size $5 \times 5 \times 48$) on the first GPU. The other half is handled by the second GPU. Note that now the depth dimension agrees. Each convolutional layer is followed by a ReLU[3] non-linearity. The second convolutional layer is followed by a ReLU and a max-pooling[4] layer. The five convolutional layers are followed by three fully connected layers.

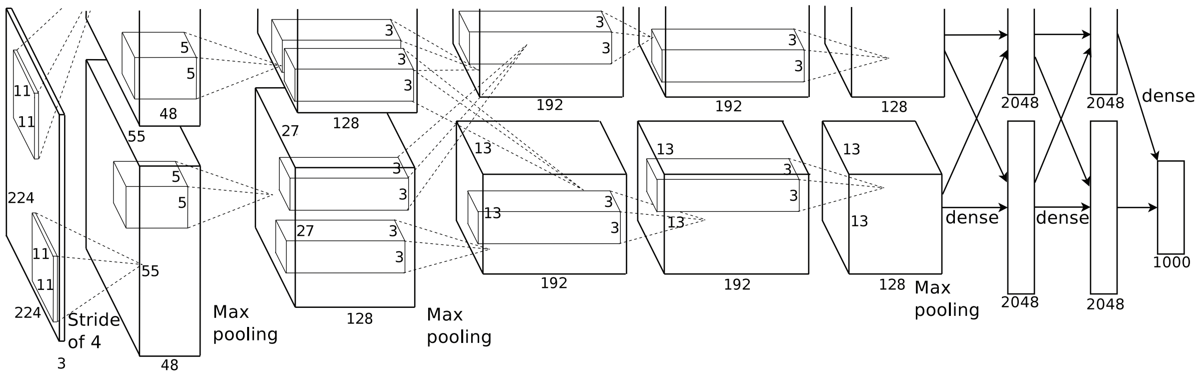


Figure 2.3.: The architecture of the winner of the ILSVRC 2012. The reference Caffe network is based on the AlexNet. (Image taken from [16])

[3]A ReLU is the abbreviation for Rectified Linear Unit. While the standard way to model a neuron's output as a function is the Sigmoid function ($f(x) = (1 + e^{-x})^{-1}$) the ReLU models a neuron's output with the function $f(x) = \max(0, x)$. CNNs with ReLU non-linearities train several times faster [16] in contrast to other non-linearities such as the Sigmoid function.

[4]A max-pooling layer reduces the spatial dimension of its input data. E.g., a max-pooling layer of 2×2 takes the maximum of each 2×2 subsection of an input and therefore reduces the dimension by a factor of 4.

The reference Caffe network is a slightly changed version of the AlexNet. E.g., an additional max-pooling layer is introduced between the first convolutional layer and the second one. The network expects a slightly bigger input size ($227 \times 227 \times 3$ vs. $224 \times 224 \times 3$). In table 2.1 all important parameters of each layer are shown. We give a definition of the symbols used:

- N: The number of images which are put into the network. We also call it the batch size. We give no value for N in the table, because we can choose the batch size freely.
- K: The number of channels a single input (image) has.
- H: The height of the input (image).
- W: The width of the input (image).
- K_h: The height of the filter. (Kernel height.)
- K_w: The width of the filter. (Kernel width.)
- g: The group parameter. Specifies the number of virtual GPUs the kernels and input data have to be split up.
- s: The stride parameter. If $s = 4$, only every 4-th pixel (across both dimensions) is convolved with the kernels.
- p_h: The number of pixels to zero-pad on both sides of the height dimension.
- p_w: The number of pixels to zero-pad on both sides of the width dimension.
- N_{output}: The number of kernels (output features).
- H_{output}: The output height.
- W_{output}: The output width.

Note that we can easily determine the output sizes of convolution layer by combining some other parameters:

$$\begin{aligned} H_{\text{output}} &= (H + 2p_h - K_h)/s + 1 \\ W_{\text{output}} &= (W + 2p_w - K_w)/s + 1 \end{aligned} \tag{2.3}$$

The data layout in Caffe is denoted by 4 dimensional vectors of size $N \times K \times H \times W$ for the input data. Filters are also stored in this data format with a size of $N_{\text{output}} \times K \times K_h \times K_w$. The output is of size $N \times N_{\text{output}} \times H_{\text{output}} \times W_{\text{output}}$. We describe the storage architecture in more detail in section 3.1.3.

Table 2.1.: Convolution layer parameters used in the reference Caffe Network

	N	K	H	W	K_h	K_w	g	s	p_h	p_w	N_{output}	H_{output}	W_{output}
conv1		3	227	227	11	11	1	4	0	0	96	55	55
conv2		96	27	27	5	5	2	1	2	2	256	27	27
conv3		256	13	13	3	3	1	1	1	1	384	13	13
conv4		384	13	13	3	3	2	1	1	1	384	13	13
conv5		384	13	13	3	3	2	1	1	1	256	13	13

2.4. Convolution in Caffe

In a CNN, many convolutions are performed in each layer. The first layer of the reference Caffe network takes images with the size of $227 \times 227 \times 3$ as input and has 96 kernels of the size $11 \times 11 \times 3$, which means that there are 96 kernels of size 11×11 for each of the 3 channels of the input image (see also table 2.1). The convolution we defined in section 2.2 is computationally very expensive. For each output of a single convolution, there are $K_h \cdot K_w$ multiplications. We perform such a convolution for every channel and sum these up over the channels. Thus, we eliminate the channel dimension in the output. The output size of a single convolution is $H_{\text{output}} \times W_{\text{output}}$. If we use the standard convolution technique there are a total number of $\#_{\cdot} = 11 \cdot 11 \cdot 55 \cdot 55 \cdot 96 \cdot 3 = 105.415.200$ multiplications and the same amount of additions ($\#_+$) per image just in the first convolution layer of the reference Caffe network.
In Caffe another method like described in [3] is used. As we can see in $\#_{\cdot}$ and $\#_+$ the number of arithmetic operations for a convolution scales quadratically with respect to the kernel size. Applying a kernel of the size 7×7 needs almost twice the number of arithmetic operations as needed for a kernel of size 5×5. Each value of an input image is used more than once during a convolution. This leads to cache unfriendly memory accesses. The method used in Caffe and [3] is called *unfolding and duplicating* of inputs. Values used more often are duplicated and unfolded into a new matrix. The kernels are also unfolded into a separate matrix. The method calculates a matrix product of the unfolded inputs and weights which yields an output matrix.
For every output feature, we compute $K = 3$ (one per input channel) convolutions. Since we only forward one output per feature to the next layer, we sum up the convolution results of each channel to obtain a single result. We can use this fact and perform a matrix multiplication to sum up the channels' convolution results. In figure 2.4 a convolution of an input image with a size of 3×3 with $K = 3$ channels is computed. For every output feature ($N_{\text{output}} = 2$) and channel of the input image there is a corresponding convolution kernel. We compute K convolutions per output feature and sum them up as seen on the top of the figure.

On the bottom of the figure, the convolution by matrix multiplication method is illustrated. For each channel, we extract subsections with the size of the convolution kernel and write each subsection into a new matrix' row, where we store channels consecutively. Note that

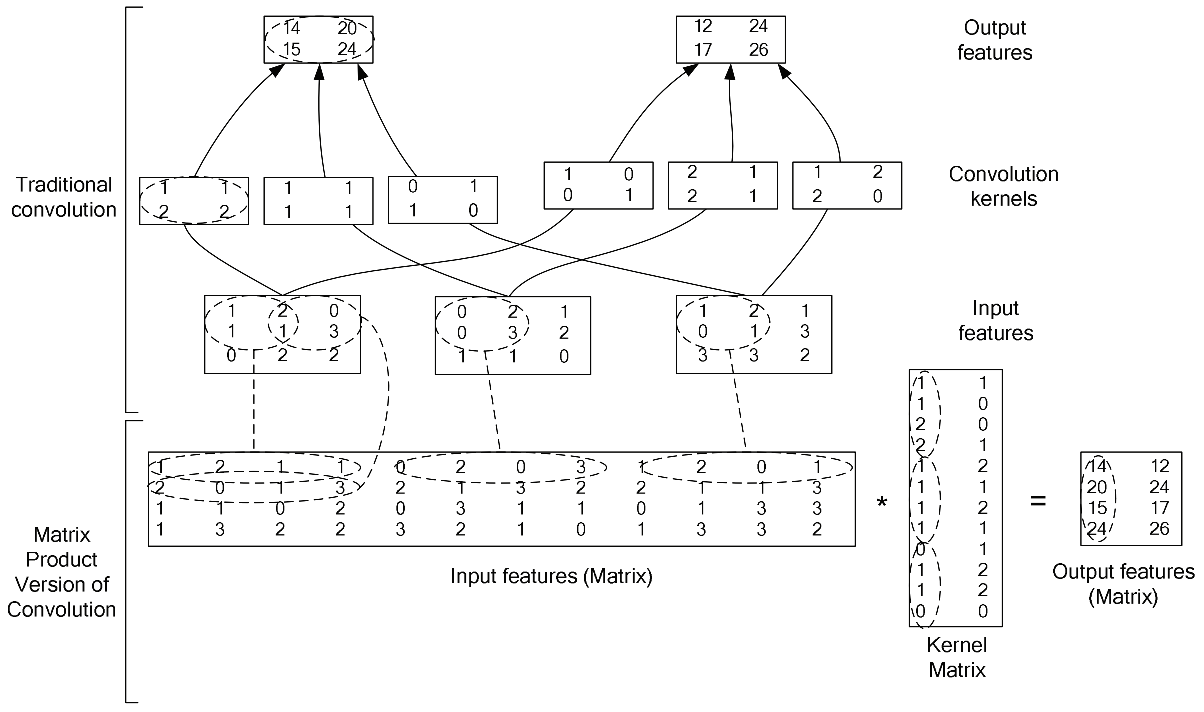


Figure 2.4.: Example of convolution by matrix multiplication. The top part of the figure shows the naive way of performing a convolution in a CNN while the bottom shows the faster matrix convolution. (Image taken from [3])

this is the unfolding we spoke of earlier. Each row stands for another output position (the output size of a feature is 2×2 which results in 4 rows). In the kernel matrix, we do not need to duplicate values and unfold a single feature's values into one column. Note that we implicitly sum up over all channels, because of the definition of a matrix multiplication. The output matrix has to be reshaped back to yield the same form as the output features seen on top of the figure. We see in the resulting matrix that one column represents one output feature.
Matrix multiplications can be computed very efficiently on modern CPUs and GPUs because of heavily tuned BLAS[5] libraries [17] like Intel's Math Kernel Library (MKL) [5] or OpenBLAS [26]. Matrix multiplications with BLAS libraries do not suffer from cache unfriendly memory accesses in contrast to the naive convolution method.
Figure 2.5 shows the unfolding of the inputs that is needed to do the matrix multiplication convolution as performed in a Caffe convolution layer. As we described in the previous example, each channel of the input features on which the kernel is applied is unfolded into a separate part of the resulting input features matrix. This matrix has a size of $H_{\text{output}} \cdot W_{\text{output}} \times K_h \cdot K_w \cdot K$, which is 3025×363 for the first convolution layer of the reference Caffe network. The kernel matrix has a size of $K_h \cdot K_w \cdot K \times N_{\text{output}}$, which is 363×96. The output features matrix's size is $H_{\text{output}} \cdot W_{\text{output}} \times N_{\text{output}}$, which is 3025×96. We exactly produce the desired output, which we forward to the next layer.

The naive multiplication algorithm for square matrices needs $\mathcal{O}(\dot{N}^3)$ arithmetic operations. However, this only applies to square matrices with a side length of $\dot{N}$. We denote a general matrix multiplication by the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$$

The naive multiplication of rectangular matrices needs $2\dot{M}\dot{N}\dot{K}$ operations, whereas $\dot{M}$ are the number of rows in matrix $\mathbf{A}$ and $\mathbf{C}$, $\dot{N}$ are the number of columns in matrix $\mathbf{B}$ and $\mathbf{C}$ and $\dot{K}$ specifies the number of columns in matrix $\mathbf{A}$ and the number of rows in matrix $\mathbf{B}$.

The complexity for the naive matrix multiplication algorithm yields $2 \cdot 3025 \cdot 96 \cdot 363 = 2 \cdot 105{,}415{,}200 = 210{,}830{,}400$ arithmetic operations for the first convolution layer (105,415,200 multiplications and 105,415,200 additions). Note that this number is the same as for the simple convolution algorithm. In 1969 Volker Strassen [24] proposed an algorithm for faster multiplication of square matrices with a complexity of $\mathcal{O}(\dot{N}^{\log_2 7}) \approx \mathcal{O}(\dot{N}^{2.807})$. This algorithm is used in the well-known BLAS libraries mentioned before. In [10] the number of multiplications for rectangular matrices using the algorithm of Strassen is derived. The number of multiplications needed is $\min(\dot{M}, \dot{N}, \dot{K})^{\log_2 \frac{7}{8}} \cdot \dot{M}\dot{N}\dot{K}$. There is no general formula for the number of additions, but it is of the same magnitude as the number of multiplications. We get a total of $2 \cdot \min(3025, 96, 363)^{\log_2 \frac{7}{8}} \cdot 3025 \cdot 96 \cdot 363 = 2 \cdot 96^{\log_2 \frac{7}{8}} \cdot 105{,}415{,}200 \approx 87{,}510{,}152$ arithmetic operations, which are 123,320,248 less than estimated for the naive algorithm. Therefore, Caffe can achieve a speed-up of 2.4×

[5]BLAS stands for *Basic Linear Algebra Subprograms*.

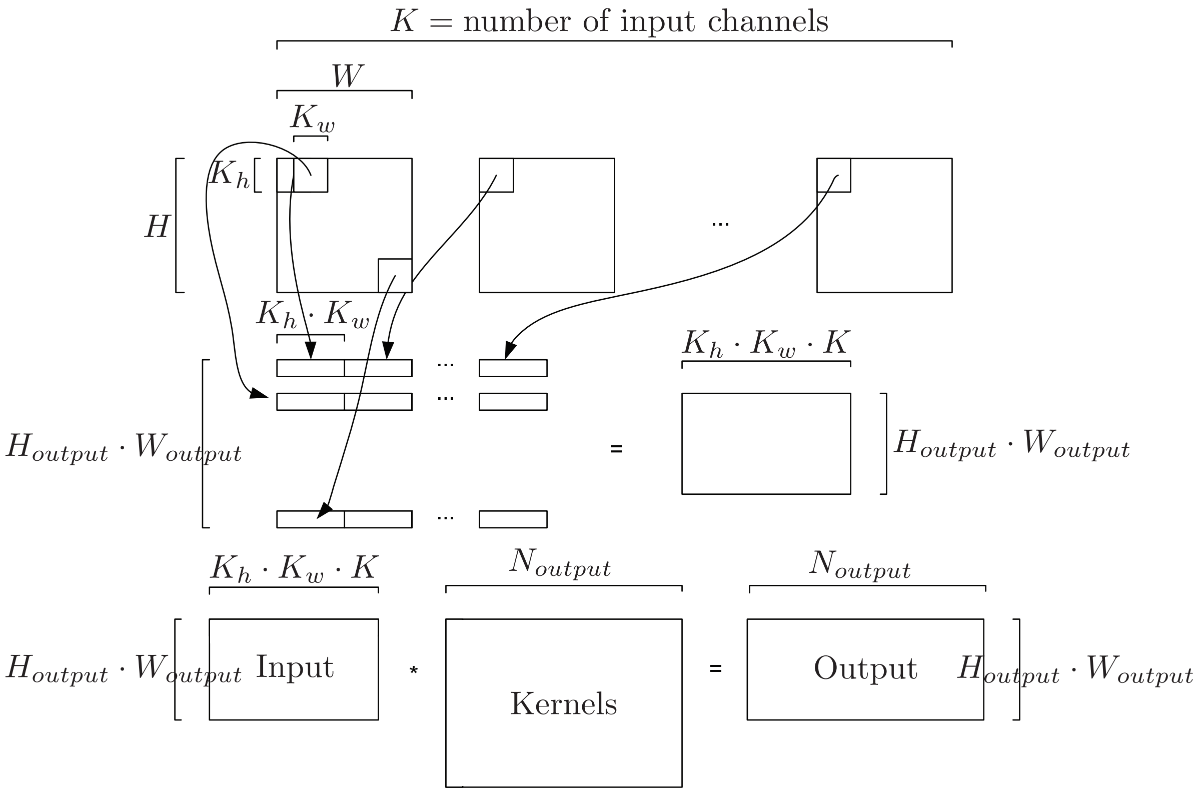

K = number of input channels
W
K_w
K_h
H
...
$K_h \cdot K_w$
$K_h \cdot K_w \cdot K$
$H_{output} \cdot W_{output}$
=
$H_{output} \cdot W_{output}$
$K_h \cdot K_w \cdot K$
N_{output}
N_{output}
$H_{output} \cdot W_{output}$
Input
*
Kernels
=
Output
$H_{output} \cdot W_{output}$

Figure 2.5.: Unfolding of inputs in a convolution layer into the matrix product convolution (Image taken from [3] and altered.)

in layer 1 by using the alternative method of convolution by matrix multiplication. In addition, memory friendly accesses speed up the convolution even more. In Caffe or MATLAB the method of unfolding before the matrix multiplication is implemented in a method called `im2col`.

2.5. Fourier Transforms

In this section, we give a short introduction to the Discrete Fourier Transform and the Fast Fourier Transform, which are powerful tools for analyzing and processing digital signals in the frequency domain rather than the time domain. After applying those transforms, we can analyze a signal with respect to the frequencies it contains rather than with respect to the time. This enables us to do powerful processing operations, which are otherwise not possible.

2.5.1. Discrete Fourier Transform

In the following section we use definitions and properties proposed by Richard Lions in [18, Chapter 3]. The foundation for the *Discrete Fourier Transform* (DFT) is the (continuous) Fourier transform. We define the continuous Fourier transform as follows:

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi tf} dn \tag{2.4}$$

In (2.4) $x(t)$ denotes a continuous time domain input signal, whose frequency contents will be decomposed into a continuous frequency domain signal $\hat{x}(f)$ by taking the Fourier transform. Every signal is composed out of frequencies, which the Fourier transform recovers. f is a frequency in the decomposed signal while $e \approx 2.71818$ is Euler's Number and $i = \sqrt{-1}$ is the imaginary unit. The Fourier transform delivers the frequency content or harmonic content of any signal.
The DFT does the same for non-continuous signals and is the mathematical method for determining the frequencies of a discrete signal sequence. We describe an input signal sequence with a vector $\mathbf{x}$ and define the DFT like this:

$$\hat{\mathbf{x}}_m = \sum_{n=0}^{N-1} \mathbf{x}_n e^{-i2\pi nm/N} \tag{2.5}$$

N is the number of samples within in the vector $\mathbf{x}$ and m is the output index. m ranges from 0 to $N-1$ for complex-to-complex transforms (complex valued inputs are assumed). $\mathbf{x}_n$ describes the n-th component of the vector $\mathbf{x}$. Each complex number in $\hat{\mathbf{x}}$ describes

the amplitude and phase angle of a frequency. The frequency of $\hat{\mathbf{x}}_m$ is $\frac{mf_s}{N}$, whereas f_s is the sampling frequency of the signal.
For real-to-complex transforms (we only use real values in our input $\mathbf{x}$), we only need to calculate $(N/2)+1$ frequency values in the output vector $\hat{\mathbf{x}}$. This follows because of the Hermitian symmetry of real-valued DFTs, which states that if we use real values as input, the frequency values from $m = 1$ to $m = (N/2) - 1$ are redundant with the frequency values of $m > (N/2)$. Each value can be constructed by its Hermitian symmetric value. The amplitude of a value and its Hermitian symmetric value are the same, while the phase angle of m is exactly the negative phase angle of $N - m$. We can describe this fact with the complex conjugate[6] as follows:

$$\hat{\mathbf{x}}_m = \overline{\hat{\mathbf{x}}_{N-m}}$$

Because of the conjugate symmetry of the DFT only the first $(N/2)+1$ values are independent and we only need to compute those values. We can easily prove the conjugate symmetry by substituting $N - m$ for m in (2.5):

$$\begin{aligned}\hat{\mathbf{x}}_{N-m} &= \sum_{n=0}^{N-1} \mathbf{x}_n e^{-i2\pi n(N-m)/N} = \sum_{n=0}^{N-1} \mathbf{x}_n e^{-i2\pi nN/N} e^{-i2\pi n-m/N} = \\ &= \sum_{n=0}^{N-1} \mathbf{x}_n e^{-i2\pi n} e^{i2\pi nm/N} = \sum_{n=0}^{N-1} \mathbf{x}_n 1 e^{i2\pi nm/N}\end{aligned} \tag{2.6}$$

We can achieve the last step in (2.6), because according to Euler's formula[7] $e^{-i2\pi n} = \cos(2\pi n) - i\sin(2\pi n) = 1$. Note that $\forall n \in \mathbb{N} : \cos(2\pi n) - i\sin(2\pi n) = 1$. We can easily see that (2.6) is the complex conjugate of (2.5).

We can now calculate the DFT for our discrete example signal $\mathbf{i} = \begin{bmatrix} 3 & 1 & 2 & 7 & 0 & 5 & 8 & 4 \end{bmatrix}$ from section 2.2. Because of the symmetry of real-to-complex transforms we only need to calculate the first $(N/2) + 1 = 5$ values for the DFT ($m \in [0; 4]$). For $m = 0$ (2.5) evaluates to $\hat{\mathbf{i}}_0 = \sum_{n=0}^{N-1} \mathbf{i}_n$ and we only need to sum up all input values:

$$\begin{aligned}\hat{\mathbf{i}}_0 &= 3 + 1 + 2 + 7 + 0 + 5 + 8 + 4 = 30 + 0i \\ \hat{\mathbf{i}}_1 &= 3 \cdot e^{-i2\pi 0 \cdot 1/8} + 1 \cdot e^{-i2\pi 1 \cdot 1/8} + 2 \cdot e^{-i2\pi 2 \cdot 1/8} + 7 \cdot e^{-i2\pi 3 \cdot 1/8} \\ &\quad + 0 \cdot e^{-i2\pi 4 \cdot 1/8} + 5 \cdot e^{-i2\pi 5 \cdot 1/8} + 8 \cdot e^{-i2\pi 6 \cdot 1/8} + 4 \cdot e^{-i2\pi 7 \cdot 1/8} = \\ &= 3 + 1 \cdot (0.7071 - 0.7071i) + 2 \cdot (6.1232 \cdot 10^{-17} - 1i) + \\ &\quad + 7 \cdot (-0.7071 - 0.7071i) + 0 + 5 \cdot (-0.7071 + 0.7071i) + \\ &\quad + 8 \cdot (-1.8370 \cdot 10^{-16} + 1i) + 4 \cdot (0.7071 + 0.7071i) = -1.9497 + 6.7071i \\ \hat{\mathbf{i}}_2 &= -7 + 5i \\ \hat{\mathbf{i}}_3 &= 7.9497 - 5.2929i\end{aligned}$$

[6]The complex conjugate of a complex number $c = a + ib$ is defined as $\overline{c} = a - ib$.
[7]Euler's formula is defined by $e^{ix} = \cos x + i \sin x$

$$\hat{\mathbf{i}}_4 = 4 + 0i$$
$$\hat{\mathbf{i}}_5 = 7.9497 + 5.2929i$$
$$\hat{\mathbf{i}}_6 = -7 - 5i$$
$$\hat{\mathbf{i}}_7 = -1.9497 - 6.7071i$$

Taking the real-to-complex DFT of our real vector therefore yields:

$$\hat{\mathbf{i}} = \begin{bmatrix} 30 & -1.9497 + 6.7071i & -7 + 5i & 7.9497 - 5.2929i & 4 \end{bmatrix}$$

There is also a transform called the Inverse Discrete Fourier Transform (IDFT), which reverses the DFT as the name suggests. The definition of the IDFT is quite simple, as it is like (2.5) but with a positive i in the exponent and normalized with $\frac{1}{N}$:

$$\mathbf{x}_n = \frac{1}{N} \sum_{m=0}^{N-1} \hat{\mathbf{x}}_m e^{i2\pi nm/N} \tag{2.7}$$

Note that if we want to get our input vector $\mathbf{i}$ back, we have to supply $\hat{\mathbf{x}}$ of length $N = 8$. We can supply this vector easily by using $\hat{\mathbf{x}}_m = \overline{\hat{\mathbf{x}}_{N-m}}$ for $m > (N/2)$ like we described before.
We can also compute the DFT of two-dimensional signals such as images. We have to expand the DFT equation from (2.5) by a second dimension:

$$\hat{\mathbf{X}}_{u,v} = \sum_{y=0}^{H-1} \sum_{x=0}^{W-1} \mathbf{X}_{y,x} e^{-i2\pi(yu/H + xv/W)} \tag{2.8}$$

u and v are the output indices of the 2D-DFT while y and x are the image's input indexes. The image size is $H \times W$ (height $\times$ width). The output of the real-to-complex 2D-DFT contains some redundancies, too. Generally, one of the dimensions in any n dimensional DFT can be reduced like shown before. Therefore, the count of complex numbers needed to store the two dimensional real-to-complex transform is

$$H \cdot (W/2) + 1. \tag{2.9}$$

Of course we can also inverse the 2D-DFT. We expand (2.7) by a second dimension and normalize by it:

$$\mathbf{X}_{y,x} = \frac{1}{HW} \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \hat{\mathbf{X}}_{u,v} e^{i2\pi(yu/H + xv/W)} \tag{2.10}$$

2.5.2. Fast Fourier Transform

As seen in the example in section 2.5.1 the DFT takes many arithmetic operations to complete. For a 8-point complex-to-complex transform we have to perform $N^2 = 64$

complex multiplications. The DFT is the most straightforward way to determine the frequencies a sequence in the time domain contains, but it is very slow as it scales quadratically with the input size.
In 1965 Cooley and Tukey [4] found a very efficient algorithm for computing the DFT commonly known as the Fast Fourier Transform (FFT) or Radix-2 FFT. The algorithm is known as the Radix-2 FFT since it *only* takes inputs with a size of $N = 2^k$ whereas k is any positive integer. We can see in our example, where we determined the frequency domain representation of the input signal, that there are many redundant operations within the evaluation of $\hat{\mathbf{x}}_1$ (Note the factor 0.7071 appearing all over the evaluation with mere sign changes). The FFT uses this fact among others to save complex multiplications. The number of complex multiplications within the FFT are approximately (because mere sign changes are not considered a multiplication) $\frac{N}{2}\log_2 N$. In our example from before we would only need $4 \cdot \log_2 8 = 12$ complex multiplications instead of 64. For a $2^{14} = 16384$-point DFT we would need about 268 million complex multiplications, whereas the FFT would only need 114,688 complex multiplications. This is an approximate speed-up of a factor 2340×. To put that in perspective, figure 2.6 shows the number of calculations (y-axis) needed for the FFT vs. the DFT with respect to the input size N shown on the x-axis.

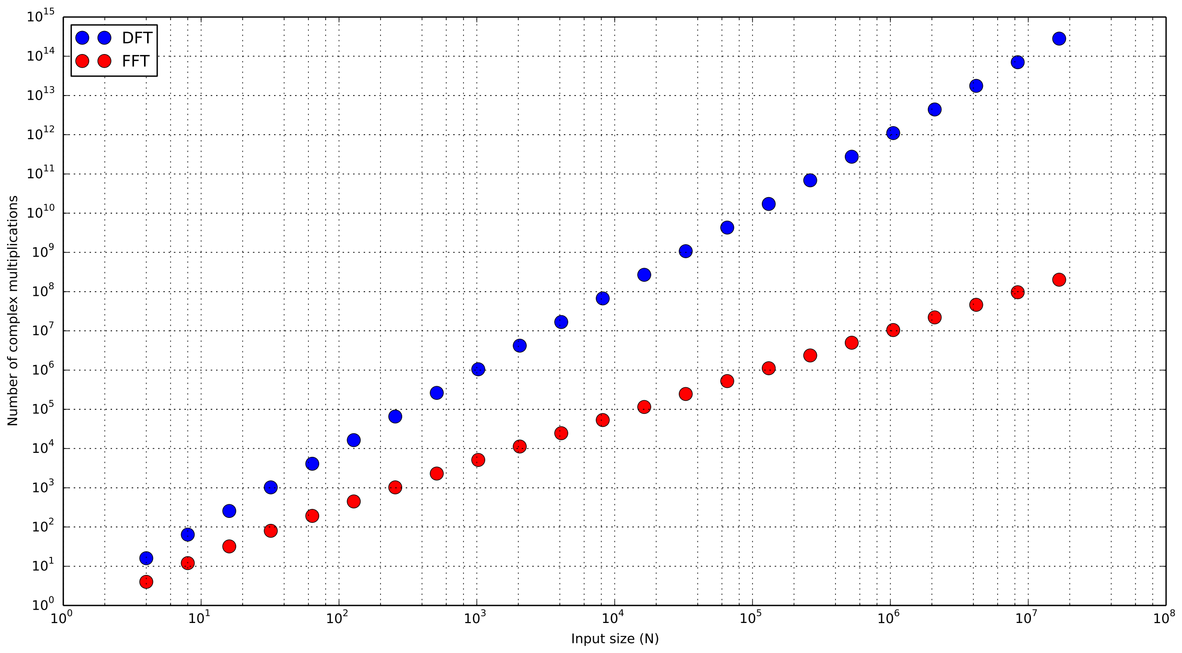


Figure 2.6.: The plot shows the number of complex multiplications needed by the DFT and FFT with respect to the input size. The DFT is shown in blue and the FFT in red.

It is important to note that the FFT is no approximation of the DFT. The FFT *is* equal to the DFT. Now we give a short derivation of the FFT algorithm. The details can also be found in [18, Chapter 4.3].

The FFT is a divide and conquer algorithm. We split the sum within the DFT (2.5) into two sums of length $N/2$, which sum up the even and odd terms for n respectively. We also extract the so-called twiddle factors ($W_N^m = e^{-i2\pi m/N}$) out of the second sum. These twiddle factors can be reused many times. After a few more steps we can represent the outputs of the DFT like this:

$$\begin{aligned} \hat{\mathbf{x}}_m &= \sum_{n=0}^{(N/2)-1} \mathbf{x}_{2n} W_{N/2}^{nm} + W_N^m \sum_{n=0}^{(N/2)-1} \mathbf{x}_{2n+1} W_{N/2}^{nm} \\ \hat{\mathbf{x}}_{m+N/2} &= \sum_{n=0}^{(N/2)-1} \mathbf{x}_{2n} W_{N/2}^{nm} - W_N^m \sum_{n=0}^{(N/2)-1} \mathbf{x}_{2n+1} W_{N/2}^{nm} \end{aligned} \tag{2.11}$$

Note that m now ranges from 0 to $(N/2) - 1$. In (2.11) we can see clearly that we only have to compute $\hat{\mathbf{x}}_m$, which saves us the reevaluation of any sine or cosine multiplications for $\hat{\mathbf{x}}_{m+N/2}$. We can simply reuse the values already calculated in $\hat{\mathbf{x}}_m$. Because we required the input size to be a power of 2 we can repeatedly split up this sum. Hence, we obtain the logarithmic runtime complexity of the FFT algorithm.
In our example from before we had an input size of $N = 8$ which is a power of two and satisfies our premise. But how can we transform input sizes other than $N = 2^k$? We can do this in two different ways. We can crop the input size to the next lower N that satisfies the premise. However, by doing this, we would lose significant samples of the input vector, which is not an option. We can also increase the length of our input vector to the next power of 2 by appending zeros at the end. This technique called zero-padding ensures that every data sample is used.
There are other Radix FFT algorithms like Radix-3 or Radix-5 which split the DFT sum up into 3 or 5 smaller DFTs respectively. Modern CPU or GPU implementations of the FFT, therefore, support input sizes of $N = 2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e$ or sizes composed out of even higher prime factors, whereas the Radix-2 FFT performs better than the Radix-3 FFT and so on. We denote the FFT compatible size of N by N_{FFT}. We can apply the Radix algorithms on the IDFT (2.7) as well. We call this the Inverse Fast Fourier Transform (IFFT). Furthermore, the FFT can be applied to the 2D-DFT and 2D-IDFT as well. The FFT sizes for the 2D-FFT are denoted by H_{FFT} and W_{FFT} respectively.

2.6. Convolution in the Frequency Domain

We already performed the convolution in the time domain (see section 2.2). But we can also compute it within the frequency domain. The **convolution theorem** states that the Fourier transform of the convolution between two functions is equal to the multiplication of their Fourier transforms:

$$\mathcal{F}(f(x) * g(x)) = \mathcal{F}(f(x)) \cdot \mathcal{F}(g(x)) \tag{2.12}$$

A detailed derivation of this theorem is given in appendix A. The derivation is given for the continuous Fourier transform but also applies to the DFT as stated in [2, p. 367].
To perform the discrete convolution in the frequency domain we take the DFT of an input signal $\mathbf{i}$ and a filter $\mathbf{f}$. By performing a pointwise multiplication (also known as Hadamard product $\circ$) between the resulting vectors we get the DFT of the convolution between $\mathbf{i}$ and $\mathbf{f}$:

$$\mathcal{F}(\mathbf{i} * \mathbf{f}) = \mathcal{F}(\mathbf{i}) \circ \mathcal{F}(\mathbf{i}) = \hat{\mathbf{i}} \circ \hat{\mathbf{f}}$$

If we take the IDFT (denoted by $\mathcal{F}^{-1}(\cdot)$) of the result we get the exact result as the convolution in the time domain yields:

$$\mathcal{F}^{-1}(\hat{\mathbf{i}} \circ \hat{\mathbf{f}}) = \mathbf{i} * \mathbf{f}$$

We computed $\mathcal{F}(\mathbf{i}) = \hat{\mathbf{i}}$ earlier (see section 2.5.1):

$$\hat{\mathbf{i}} = \begin{bmatrix} 30 & -1.9497 + 6.7071i & -7 + 5i & 7.9497 - 5.2929i & 4 \end{bmatrix}$$

A pointwise multiplication between $\hat{\mathbf{i}}$ and $\hat{\mathbf{f}}$ is only possible if the vectors have the same length. Therefore we have to zero-pad $\mathbf{f}$ to length $N_{\text{FFT}} = 8$ before taking the DFT. Note that N_{FFT} is the FFT friendly size of $\max(\dim(\mathbf{i}), \dim(\mathbf{f}))$ for both the input and the filter. The DFT of $\mathbf{f}$ evaluates to:

$$\hat{\mathbf{f}} = \begin{bmatrix} 0 & 1 + 1i & 2 & 1 - 1i & 0 \end{bmatrix}$$

We compute the pointwise multiplication between $\hat{\mathbf{i}} \circ \hat{\mathbf{f}}$ and the IDFT of the result:

$$\begin{aligned} \hat{\mathbf{i}} \circ \hat{\mathbf{f}} &= \begin{bmatrix} 0 & -8.6569 + 4.7574i & -14 + 10i & 2.6569 - 13.2426i & 0 \end{bmatrix} \\ \mathcal{F}^{-1}(\hat{\mathbf{i}} \circ \hat{\mathbf{f}}) &= \begin{bmatrix} -5 & -3 & -1 & 6 & -2 & -2 & 8 & -1 \end{bmatrix} \end{aligned}$$

We see that after taking the IDFT, we obtain the *same* result we computed in section 2.2. The first two values are a result of the circular convolution. In the example from section 2.2 we computed the convolution from position 1 through 6. The circular convolution simply overlaps to the end or the start respectively when performed at position 0 and 7.
Because we can perform the DFT very fast by using the FFT, the convolution in the frequency domain can accomplish considerable speed-ups in contrast to the time domain convolution. Again, we can do the same thing for the two dimensional case. The IFFT of the pointwise product between the Fourier Transforms of two matrices is equal to their convolution in time space:

$$\mathcal{F}^{-1}(\hat{\mathbf{I}} \circ \hat{\mathbf{K}}) = \mathbf{I} * \mathbf{K}$$

We define the FFT friendly sizes for the two-dimensional transform similar to the one-dimensional transform:

$$\begin{aligned} H_{\text{FFT}} &= \text{next compatible prime size of} \max(H, K_h) \\ W_{\text{FFT}} &= \text{next compatible prime size of} \max(W, K_w) \end{aligned}$$

We present details on the methodology for obtaining the next compatible prime size in section 3.3.3.

2.7. Fast pointwise multiplication

We can use the fact that the FFT is linear to further improve the performance. Remember that in a convolutional layer within a neural network 2D convolution results of each channel are summed up before the results are forwarded to the next layer. In the first layer of our reference network, we perform 96 convolutions for each of the 3 channels. Because the FFT is linear, we can sum up the results over the channels and only take the IFFT of 96 results instead of 288.
In 2015 Vasilache et al. proposed a novel way [25] to reduce the problem of the pointwise multiplication to a matrix multiplication. As we explored before matrix multiplications within famous BLAS libraries are heavily tuned and perform many times faster because they use the algorithm proposed by Strassen. In figure 2.7 both versions of the pointwise multiplications are demonstrated. The simple pointwise multiplication as we described in section 2.6 is on the left side of the figure. Assume we have a batch size of 2 images with 2 channels passed to our convolutional layer. The 2 images are separated by a vertical line in the figure and the channels are next to each other. Assume the 2D-FFT of each channel in each image yields a 2×2 frequency domain representation. Therefore we have a complex input sequence $\hat{\mathbf{I}}$ of size $2 \times 2 \times 2 \times 2$. Note that we not have used the imaginary unit i in the figure for simplicity reasons. Also, we have a frequency domain representation $\hat{\mathbf{K}}$ of a single kernel that has the same channel dimension as the input (1 kernel with 2 channels of size 2×2). If we pointwise multiply the first image with the kernel and sum up the results of the channels, we obtain the output for the first image. We handle the second image accordingly. The results are shown on the bottom left of the figure ($\hat{\mathbf{O}}$).

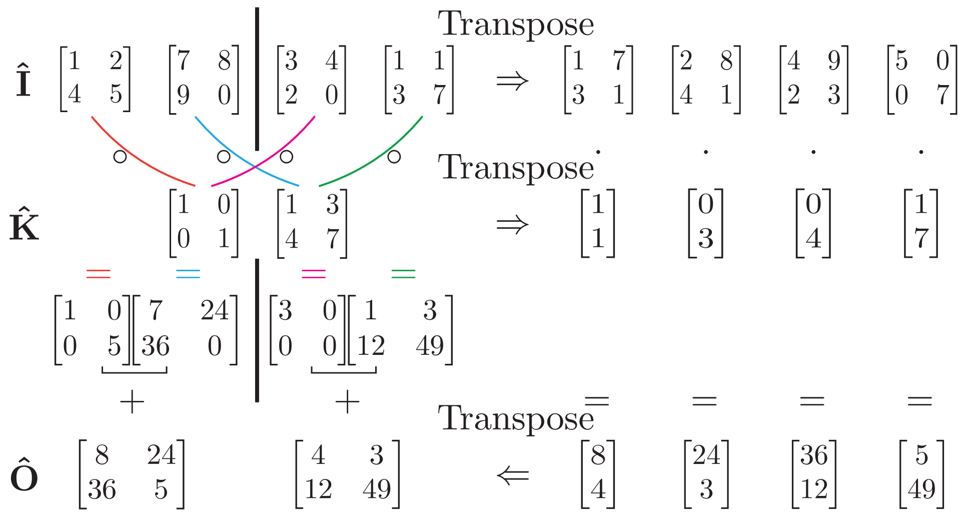


Figure 2.7.: Two ways of pointwise multiplication in the frequency domain. On the left the naive method is shown. After the pointwise multiplication values have to be summed up to get the result. On the right side the matrix multiplication version is shown.

Remember the data layout notation from section 2.3 which is denoted by $N \times K \times H_{\text{FFT}} \times W_{\text{FFT}}$. For the input, N equals the number of images passed to the convolutional layer while for the kernel $N = N_{\text{output}}$. We transpose the inputs and kernels to a $H_{\text{FFT}} \times W_{\text{FFT}} \times N \times K$ layout (as we can see on the right side of the figure) to perform the matrix multiplication. Afterwards, we perform $H_{\text{FFT}} \cdot W_{\text{FFT}} = 4$ matrix multiplications with matrices of sizes $N \times K$ and $(N_{\text{output}} \times K)^{\mathsf{T}}$. Note that K is the same for each transposed sequence and we therefore can perform a matrix multiplication. The matrix multiplications yields $H \cdot W = 4$ matrices of size $N \times N_{\text{output}}$ each. We can see the result of size $2 \times 2 \times 2 \times 1$ on the bottom right. If we transpose the result back, we get an output sequence of size $2 \times 1 \times 2 \times 2$, which yields the exact same result as the simple pointwise multiplication. As we discovered at the beginning of this section we can now take the IFFT of the already channel-reduced output $\hat{\mathbf{O}}$ to get a real-valued result of the convolution operation as performed in a convolutional layer.

2.8. Theoretical speed-ups

In the following section we explore the theoretical speed-ups we could accomplish by replacing the Caffe convolution (see section 2.4) through an FFT based convolution. We exemplary compute the FLOP[8] count on the reference Caffe network to get an estimate if the FFT convolution can pay off in contrast to the Caffe convolution. We calculated our estimate for the Caffe convolution for every layer. We denote the Caffe convolution by `caffe-conv` and # as abbreviation for the FLOP count.

$$\begin{aligned}
m := \frac{N_{\text{output}}}{g}, n := H_{\text{output}} \cdot W_{\text{output}}, k := \frac{K K_h K_w}{g} \\
\#(\texttt{caffe-conv}) &= \#(\texttt{im2col}) + \#(\texttt{gemm}(m, n, k)) \cdot g \\
\#(\texttt{im2col}) &= \frac{H + 2p_h - K_h}{s + 1} \frac{W + 2p_w - K_w}{s + 1} K K_h K_w \\
\#(\texttt{gemm}(m, n, k)) &= 2 \cdot \min(m, n, k)^{\log_2(\frac{7}{8})} \cdot mnk
\end{aligned}$$

#(`caffe-conv`) is the FLOP count needed for the Caffe convolution per layer. Note that #(`im2col`) is FLOP count for reorganizing the data into the input matrix. We calculated this FLOP count based on the original implementation in the Caffe framework. #(`gemm`(m, n, k)) is the FLOP count for the general matrix multiplication (`gemm`). m, n, k are the sizes of the two matrices multiplied by each other like we explored before. All other variables are like defined in section 2.3.

[8]FLOP means floating operation. Note that this is different from FLOPS which is used to describe the number of floating operations per second.

2.8.1. Speedup through FFT

We can also estimate the FLOP count needed for the FFT convolution. We have already stated the complexity for a one dimensional FFT in section 2.5.2. We can easily expand this to the two dimensional transform and therefore also compute a FLOP count:

$$\#(\texttt{fft}(k, H_{\text{FFT}}, W_{\text{FFT}})) = kH_{\text{FFT}}W_{\text{FFT}} \log_2(H_{\text{FFT}}W_{\text{FFT}})$$

The popular FFTW [8] library can compute a real-to-complex transform with $k = 2.5$. $H_{\text{FFT}}, W_{\text{FFT}}$ are the height and width of the FFT respectively. We have to compute many FFTs (for a single input image K FFTs and N_{output} IFFTs). Also note that we do not count the FLOP count of taking the FFT of the kernels, since we only have to do this once. We can reuse the frequency representation of kernels for every pass. We have to normalize the values after taking the IFFT by $\frac{1}{H_{\text{FFT}}W_{\text{FFT}}}$ as we also described in the 2D-IDFT equation (2.10). The FLOP count for the FFT convolution can be calculated as follows:

$$\begin{aligned}
\#(\texttt{fft-conv}) &= \#(\texttt{fft-input}) + \#(\texttt{pw-simple}) + \#(\texttt{ifft}) + \#(\texttt{normalize}) \\
\#(\texttt{fft-input}) &= \#(\texttt{fft}(k, H_{\text{FFT}}, W_{\text{FFT}}/2 + 1)) \cdot K \\
\#(\texttt{pw-simple}) &= N_{\text{output}} \cdot \frac{c}{g} \cdot H_{\text{FFT}} \cdot (W_{\text{FFT}}/2 + 1) \cdot 8 \\
\#(\texttt{ifft}) &= \#(\texttt{fft}(k, H_{\text{FFT}}, W_{\text{FFT}}/2 + 1)) \cdot N_{\text{output}} \\
\#(\texttt{normalize}) &= N_{\text{output}} \cdot H_{\text{output}} \cdot W_{\text{output}}
\end{aligned}$$

$\#(\texttt{fft-input})$ is the FLOP count for taking the FFT of the input data. We need a FLOP count of $\#(\texttt{pw-simple})$ to compute the pointwise multiplication between the input and kernels. We need 6 floating operations for a single complex multiplication[9] and 2 floating operations to aggregate this result up over the channels, hence the multiplication by 8.

Figure 2.8 shows an estimation for the FLOP count of `fft-conv` and `caffe-conv` in the reference Caffe network. We can see that the first layer clearly slows down and the second layer can perform the convolution faster.

2.8.2. Speedup through FFT with fast pointwise multiplication

We can compute the FLOP count for the fast pointwise multiplication as described in section 2.7 quite simply. The only thing that differs from `fft-conv` is the FLOP count

[9]The complex multiplication between two complex numbers $(a+bi)\cdot(c+di)$ is defined as $(a+bi)\cdot(c+di) = (ac - bd) + (bc + ad)i$.

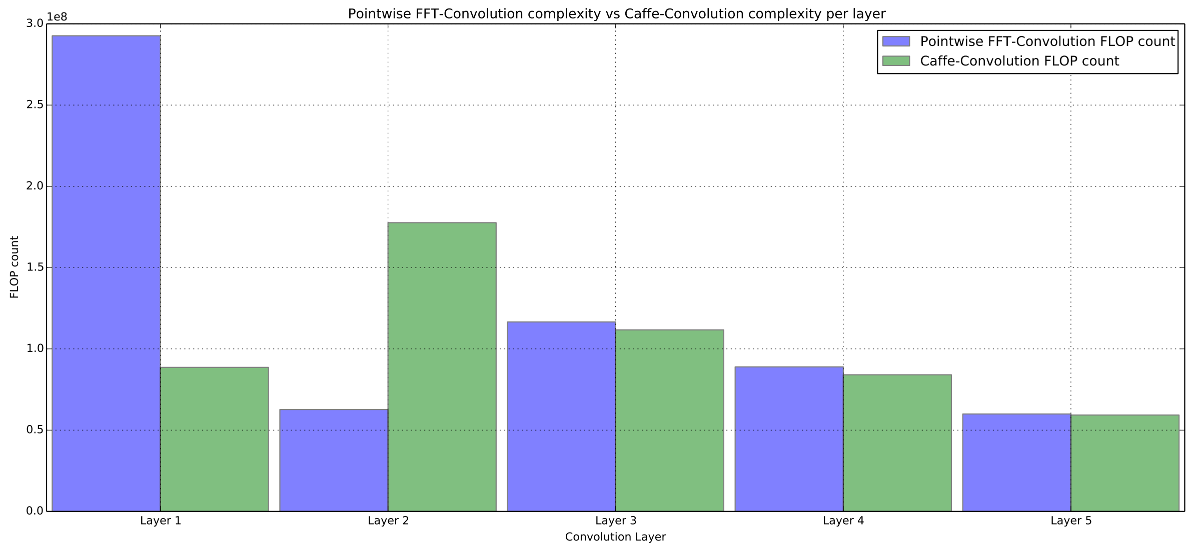


Figure 2.8.: Estimation for the FLOP count for the convolution layers within the reference network. `fft-conv` is shown in blue while `caffe-conv` is shown in green. The x-Axis shows each layer of the reference network and the y-Axis the estimated FLOP count.

of the pointwise multiplication operation. We call this method `fft-cgemm-conv` because of the complex general matrix multiplication (`cgemm`) we perform:

$$m := \text{batch size}, n := N_{\text{output}}, k := \frac{c}{g}$$

$$\#(\texttt{fft-cgemm-conv}) = \#(\texttt{fft-input}) + \frac{\#(\texttt{pw-cgemm})}{m} + \#(\texttt{ifft}) + \#(\texttt{normalize})$$

$$\#(\texttt{pw-cgemm}) = (H_{\text{FFT}} \cdot (W_{\text{FFT}}/2 + 1) \cdot g \cdot \#(\texttt{cgemm}(m, n, k)))$$

$$\#(\texttt{cgemm}(m, n, k)) = 8 \cdot \min(m, n, k)^{\log_2(\frac{7}{8})} \cdot mnk$$

Note that `cgemm` has a higher FLOP count than `gemm`, because we perform complex multiplications in contrast to `gemm` from before. We already saw in 2.7 that the fast pointwise multiplication takes advantage of the number of images processed in a single batch. Because of that we set m equal to the batch size and normalize the FLOP count of #(`pw-cgemm`) by m again to be able to compare the total FLOP count to those determined before. For our following figures we used a batch size of 500. In figure 2.9 a direct comparison between the two different versions of the pointwise multiplications is presented. We see that the estimated FLOP count of `pw-cgemm` consistently outperforms `pw-simple` in every layer.

In figure 2.10 we give the FLOP count estimation for the three different ways to perform the convolution. In contrast to figure 2.8 we see that `fft-cgemm-conv` outperforms the standard Caffe convolution in all layers except the first one. Note the significant theoretical speed-up possible in layers two and three.

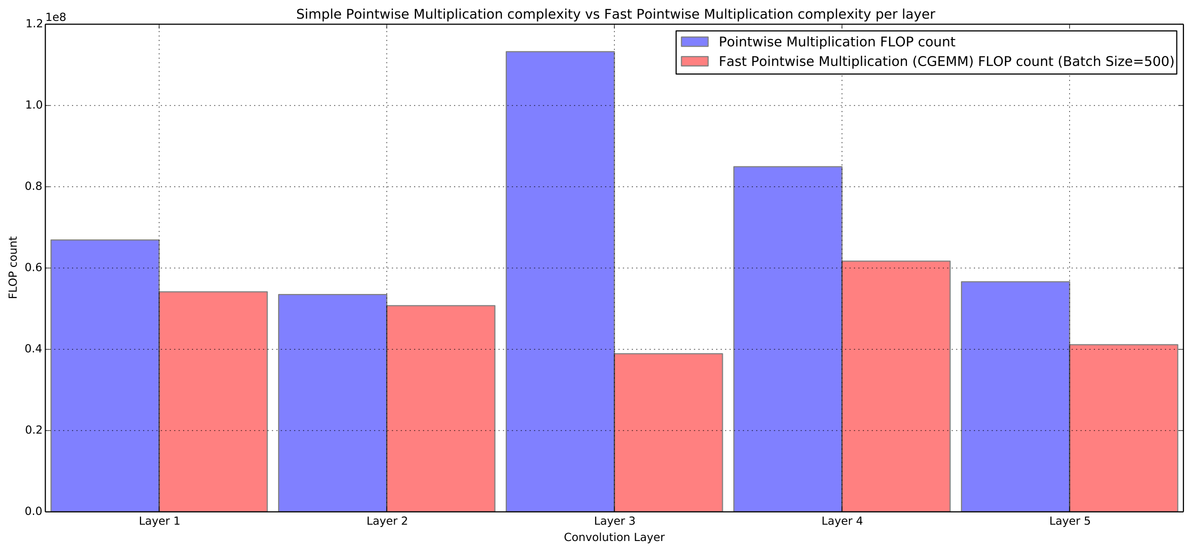


Figure 2.9.: Estimation for the FLOP count for the convolution layers within the reference network. `pw-simple` is shown in blue while `pw-cgemm` is shown in red.

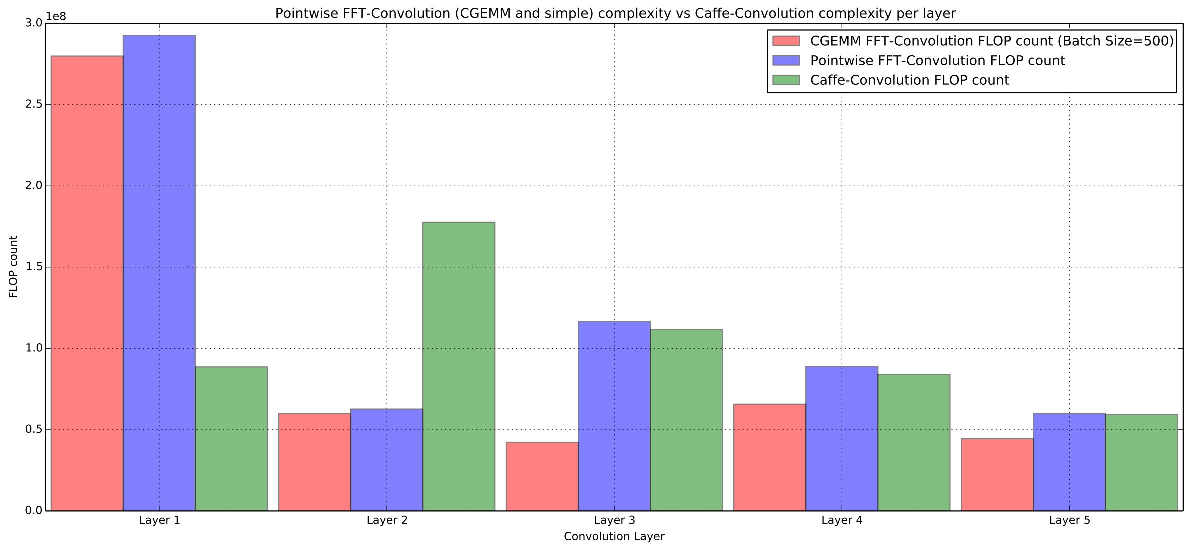


Figure 2.10.: Estimation for the FLOP count for the convolution layers within the reference Caffe network. `fft-cgemm-conv` is shown in red, `fft-conv` in blue and `caffe-conv` in red.

2.9. Backpropagation

We already explored the possibility of replacing the standard Caffe convolution method by the FFT convolution within the forward pass in a CNN. Before we can put an image into a CNN and perform the forward pass, kernels within the CNN have to be trained. For neural networks, we do this with the well-known backpropagation algorithm.

In a CNN implemented in the Caffe framework, data is forwarded from bottom to top. A layer receives its input data from a bottom vector ($\mathbf{B}$) and writes its results into the top vector ($\mathbf{T}$). We note that Caffe performs a cross-correlation[10] instead of a convolution. In a convolution layer in Caffe we have N_{output} kernels ($\mathbf{K}$) per layer. We denote each kernel's dimension again with K_h and K_w. We can alter (2.2) to fit Caffe's definitions and define the 2D cross-correlation operation as performed for a single kernel in a convolution layer. We have also omitted the depth dimension (K) of $\mathbf{B}$ and $\mathbf{K}$ in the following definition to simplify it:

$$\mathbf{T}_{y,x} = \sum_{h=0}^{K_h-1} \sum_{w=0}^{K_w-1} \mathbf{B}_{(y+h),(x+w)} \cdot \mathbf{K}_{h,w} \tag{2.13}$$

Thus, we get the entire resulting vector for an output feature by $\mathbf{T} = \mathbf{B} \star \mathbf{K}$. Note that the cross-correlation operator ($\star$) is different to the convolution operator ($*$).

2.9.1. Derivation

The following derivation is based on ideas and notations from [1, 9]. In [19] similar results were stated. Assume we have some error function E and we know the error values at a certain convolutional layer. The backpropagation algorithm updates the weights of each layer in such a way that we minimize our error function E. In Caffe the weights consist out of the kernels $\mathbf{K}$ we defined before. We denote an error with respect to some blob $\mathbf{A}$ by $\nabla\mathbf{A}$. We call this notation a gradient. We want to know the error with respect to the kernel at each position i and j of the kernel:

$$(\nabla\mathbf{K})_{i,j} = \frac{\partial E}{\partial \mathbf{K}_{i,j}} = \sum_{y=0}^{H-K_h} \sum_{x=0}^{W-K_w} \frac{\partial E}{\partial \mathbf{T}_{y,x}} \frac{\partial \mathbf{T}_{y,x}}{\partial \mathbf{K}_{i,j}} = \sum_{y=0}^{H-K_h} \sum_{x=0}^{W-K_w} \frac{\partial E}{\partial \mathbf{T}_{y,x}} \mathbf{B}_{(y+i),(x+j)} \tag{2.14}$$

Note that in (2.14) we made use of the chain rule and we therefore have to sum up over all $\mathbf{T}_{y,x}$ our kernel contributed to during the forward pass. The resulting size of $\mathbf{T}$ in the forward was $H - K_h \times W - K_w$. By looking at (2.13) we can easily follow $\frac{\partial \mathbf{T}_{y,x}}{\partial \mathbf{K}_{i,j}} = \mathbf{B}_{(y+i),(x+j)}$. We see that this looks like a cross-correlation between the bottom

[10]In the cross-correlation operation the kernel is not flipped before calculating the weighted sum.

data and the error with respect to the top data ($\nabla\mathbf{T}$). Note that $\nabla\mathbf{T}$ is something we already know:

$$\begin{aligned}(\nabla\mathbf{K})_{i,j} = \frac{\partial E}{\partial \mathbf{K}_{i,j}} &= \sum_{y=0}^{H-K_h}\sum_{x=0}^{W-K_w} \frac{\partial E}{\partial \mathbf{T}_{y,x}} \mathbf{B}_{(y+i),(x+j)} = \\ &= \sum_{y=0}^{H-K_h}\sum_{x=0}^{W-K_w} \mathbf{B}_{(y+i),(x+j)} \frac{\partial E}{\partial \mathbf{T}_{y,x}} \\ \Rightarrow \nabla\mathbf{K} = \mathbf{B} \star \nabla\mathbf{T} & \end{aligned} \tag{2.15}$$

We can now update our kernels by $\mathbf{K} = \mathbf{K} + \eta\nabla\mathbf{K}$ with η being the learning rate. We also need to propagate the error back to the previous layer. Note we perform the backpropagation algorithm and therefore the previous layer is the bottom layer. Thus, we need to calculate the error with respect to the bottom data (∇B) at every position x and y. We apply the chain rule once more and sum up over every $\mathbf{T}$ which $\mathbf{B}_{y,x}$ contributes to:

$$\begin{aligned}(\nabla\mathbf{B})_{y,x} = \frac{\partial E}{\partial \mathbf{B}_{y,x}} &= \sum_{h=0}^{K_h}\sum_{w=0}^{K_w} \frac{\partial E}{\partial \mathbf{T}_{(y-h),(x-w)}} \frac{\partial \mathbf{T}_{(y-h),(x-w)}}{\partial \mathbf{B}_{y,x}} = \\ &= \sum_{h=0}^{K_h}\sum_{w=0}^{K_w} \frac{\partial E}{\partial \mathbf{T}_{(y-h),(x-w)}} \cdot \mathbf{K}_{h,w} \\ \Rightarrow \nabla\mathbf{B} = \nabla\mathbf{T} \star \mathbf{K}^{\mathsf{T}} = \nabla\mathbf{T} * \mathbf{K} & \end{aligned} \tag{2.16}$$

By looking at (2.13) we see that $\frac{\partial \mathbf{T}_{(y-h),(x-w)}}{\partial \mathbf{B}_{y,x}} = \mathbf{K}_{h,w}$. Also note that in $\mathbf{T}_{(y-h),(x-w)}$ h and w are subtracted rather than added. We can interpret this by simply flipping the kernel. We also know that a cross-correlation with a flipped kernel is nothing other than a convolution.

2.9.2. Training with FFT

In the previous section, we derived the backpropagation algorithm for a convolution layer in Caffe. Thus, we can also implement the backward step in Caffe with the FFT methodology. We simply have to replace the cross-correlation and convolution with the FFT version. Note that we can very easily do a cross-correlation in the frequency domain by pointwise multiplying with the complex conjugate. This property is called cross-correlation theorem and described in [2, p. 368]. Note the big size of the kernel within the convolution described in (2.15) that could speed up the FFT training significantly.

3. Implementation in Caffe

In this chapter, we give some insights in our implementation of the convolution layer using the FFT convolution. We begin with a short overview of the Caffe basics needed to understand our implementation. We then go through the CPU version of a fast FFT convolution. Afterward, we explain some of the tunings we did both on the CPU and GPU to yield better performance in ways of saving arithmetic operations and memory. Finally, we illustrate differences between the GPU implementation in contrast and the CPU version.

3.1. Caffe

Caffe is implemented in C++ with interfaces built for MATLAB and Python. The code is designed for extensibility and speed. The speed is defined as how many images can be processed per second with a neural network built in Caffe. Caffe is an open source project, which enabled us to extend Caffe for our needs easily.

3.1.1. Layers and Layer Factory

Caffe contains a `Net` class which holds layers. A layer is represented by the `Layer` class. In Caffe, many different layers are implemented. Every one of those layers is a subclass of the `Layer` or an intermediate class like `BaseConvolutionLayer`. We give some layers needed by a DCNN as an example:

- `DataLayer`: The data layer provides inputs for a network. It reads images from a database and forwards them to following layers.
- `ConvolutionLayer`: Implements the convolution within a neural network. We give details of the forwarding process and convolution in section 3.2.
- `PoolingLayer`: Implements pooling in a neural network. E.g., a max-pooling layer with size of 2×2 takes the maximum out of every 2×2 portion of an input and forwards the result to the next layer.

- `ReLULayer`: In Caffe activation functions are implemented in an additional layer instead of being part of a fully connected layer or a convolution layer. `ReLULayer` implements the $f(x) = \max(0, x)$ non-linearity.
- `InnerProductLayer`: Implements a fully connected layer in a neural network. This layer treats the input as a simple vector and outputs a simple vector. Note that the input's height and width dimensions are reduced to 1.

All layers are instantiated with methods dedicated for each layer type in the so-called `layer_factory`. Here, parameters important for the layer are considered while creating a layer instance.

3.1.2. Protocol buffers

We can model Network architectures in Caffe with Google's Protocol Buffers (`protobuf`) library [11], which is used by Caffe. We can define the reference Caffe network with a `prototxt` file:

Listing 3.1: Definition of the CaffeNet with Protocol Buffers

```
name: "CaffeNet"
input: "data"
input_shape {
  dim: 10
  dim: 3
  dim: 227
  dim: 227
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
  }
}
// ...
```

Note that we only show the definitions of the first convolution layer in listing 3.1 and the input data dimensions. We expect 10 images (our batch size) with 3 channels of size 227 × 227 as input. This input is considered the bottom data for the first layer named `conv1`. We see that this layer is of type `Convolution`. The type determines the method to be called from the `layer_factory`. Within the `convolution_param` section we specify parameters for the convolution layer. We only need to specify the number of output features, the kernel size and the stride. Parameters such as padding have a default value (e.g. 0 for padding and 1 for stride) whereas the output sizes can be determined from the input size and the parameters specified. We explain the convolution parameters in more detail in section 3.2.2. The rest of the network is defined as stated in table 2.1. In a pre-trained network, Caffe loads the weights from a corresponding file which contains the weights, which are stored as defined in the protocol buffer file. If we want to train a new network, Caffe creates this corresponding weight file.

3.1.3. Memory Layout and Data Storage

Caffe stores data in its own **blob** data type. This data type can easily transfer data between the CPU and GPU. For the convolution layer, the blob data type is used for the bottom, the weights and the top amongst others. Remember that we introduced the four-dimensional nature of the convolution layer in section 2.3. On the top of figure 3.1, we see an input with a batch size of 2 images with 3 channels of size 4 × 4. Note the vertical line that divides the two images.

$$\text{Input:}\quad \begin{bmatrix}7&8&7&7\\2&2&4&8\\6&6&7&8\\1&6&6&2\end{bmatrix}\begin{bmatrix}2&8&1&7\\3&1&3&3\\4&8&8&7\\4&1&6&4\end{bmatrix}\begin{bmatrix}4&2&5&7\\4&2&3&2\\4&8&6&3\\8&8&7&4\end{bmatrix}\Bigg\|\begin{bmatrix}3&2&1&2\\0&3&8&2\\1&0&1&4\\2&0&0&7\end{bmatrix}\begin{bmatrix}4&6&5&6\\3&0&0&4\\2&5&0&2\\2&7&3&6\end{bmatrix}\begin{bmatrix}3&0&3&0\\7&3&1&6\\1&8&3&4\\7&1&6&7\end{bmatrix}$$

Memory: [7877|22...|2817...|4257...|3212|03...|4656...|3030|7316|1834|7167]

Figure 3.1.: The memory layout of a blob in Caffe. On the top the interpretation of the raw data is shown, whereas on the bottom the memory layout is visualized.

On the bottom of the figure, we see the same data as it is stored in the memory. Each two-dimensional channel of an image is stored in row-major order, whereas the channels lie in memory consecutively. The other images are stored just behind the first one in the same manner. This time, the vertical lines help to distinguish between single rows. We can access each element in a blob like this:

$$\text{The Element } (n, k, h, w) \text{ is located at index: } ((n \cdot K + k) \cdot H + h) \cdot W + w \qquad (3.1)$$

This formula applies to general 4-D blobs of size $N \times K \times H \times W$. The size of the bottom blob inside a Caffe convolution layer is $N \times K \times H \times W$, whereas the size for the weight blob is different. As we explained before we have a weight dimensionality of $N_{\text{output}} \times K \times K_h \times K_w$, thus, we have $N_{\text{output}} \cdot K$ two-dimensional kernels of size $K_h \times K_w$ each.

3.2. Convolution Layer in Caffe

In Caffe, a layer is part of a directed acyclic graph which processes data from bottom to top. I.e., we put images in the bottom of a network and we get a result at the top of a network. In this section, we describe the characteristics of the convolution layer implemented in Caffe. We emphasize that a Caffe convolution layer actually performs a cross-correlation.

3.2.1. Mechanics of the Caffe Convolution Layer

A convolution layer gets a two-dimensional image with K channels as input from the layer preceding it. We are handed this blob in the forward pass through a bottom variable. In every layer, we can use this blob and modify it any way we want. Note that in the forward pass we can only modify the top variable's content which the network automatically hands over to the subsequent layer. So we are forced to store the modifications in the top variable. In the Caffe implementation of the convolution layer, the top variable's size is determined based on the input size and the layer parameters. We already explained how to calculate the output size of a single output feature in equation (2.3). The overall sizes of the bottom and top variable evaluate to:

$$\begin{aligned} \texttt{bottom\_size} &= N \cdot K \cdot H \cdot W \\ \texttt{top\_size} &= N \cdot N_{\text{output}} \cdot H_{\text{output}} \cdot W_{\text{output}} \end{aligned}$$

We now explain the calculation steps for a single image out of the batch of N images. We know that a single input image has K channels and we have N_{output} kernels with K channels each. The Caffe convolution layer performs K two-dimensional convolutions per output feature and sums their results up into a single result, therefore, yielding the N_{output} output features. Note that if we have configured our convolution layer's stride to be greater than 1, the layer evaluates the convolution formula (2.13) at every s-th x and y of the bottom blob.

In figure 3.2 we visualize the mechanics of the Caffe convolution layer. We only explain fundamental principles of a convolution layer not the actual convolution as performed in a Caffe convolution layer. We already explored the convolution by matrix multiplication technique in section 2.4. This technique is not visualized in the figure. For the figure, we choose to use the sizes of the first layer of the reference Caffe network. On the left, a single image is the bottom variable of the forward pass. We compute the convolution between every of the 3 channels and the corresponding channel kernel of a single weight and sum up the results over the channels. We repeat this for all of the $N_{\text{output}} = 96$ weights. Thus, we get 96 output features of size 55×55. We write this result into the top variable. This top blob is passed to subsequent layers as the bottom blob. Note that the mechanics of the convolution layer are slightly different for layers that have the group parameter set to a value greater than 1. We dive into those mechanics in section 3.2.2.

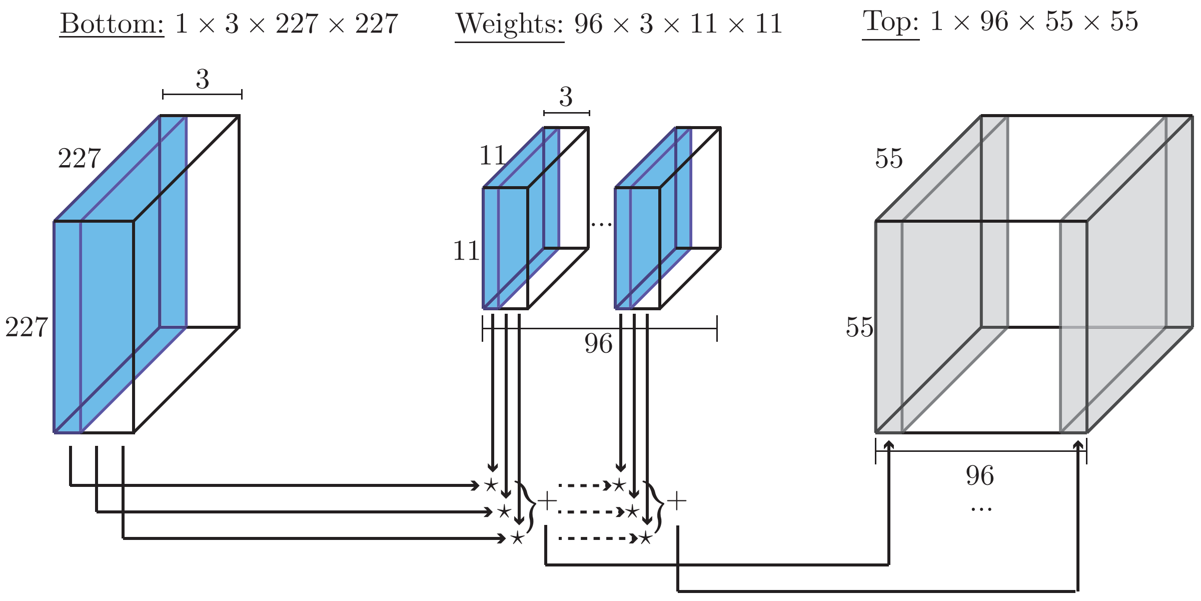

Bottom: $1 \times 3 \times 227 \times 227$
Weights: $96 \times 3 \times 11 \times 11$
Top: $1 \times 96 \times 55 \times 55$
3
227
227
3
11
11
...
96
55
55
96
...
+
+

Figure 3.2.: The convolution as performed in the first layer of the reference Caffe network in the Caffe convolution layer. Convolutions are performed for every output feature with the same bottom and forwarded to the top blob.

3.2.2. Convolution Parameters

We now give a short overview of the parameters which we can set in a convolution layer and how they affect the mechanics of the convolution layer. We have to understand the consequences these parameters can have to properly implement the FFT convolution layer in section 3.3.

kernel_size The size of a single kernel affects the output size of the top blob via the border pixels. A greater kernel size also increases the computational costs of a single convolution. Both the kernel's height and width are set to the kernel size.

num_output The number of output features determines the number of output features and corresponding weights, which will be trained for this layer. Each weight consists of

K 2D kernels. The greater this number, the more convolutions have to be computed, but more output features can also increase the prediction accuracy of the neural network.

pad We need to zero-pad the bottom blob before performing the convolution. Note that Caffe performs the convolution differently and we are only visualizing the effect of the parameters here. If we pad a single channel of the bottom blob with pad set to 2, we transform the channel's matrix representation like this:

$$\begin{bmatrix} 7 & 8 & 7 & 7 \\ 2 & 2 & 4 & 8 \\ 6 & 6 & 7 & 8 \\ 1 & 6 & 6 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 8 & 7 & 7 & 0 & 0 \\ 0 & 0 & 2 & 2 & 4 & 8 & 0 & 0 \\ 0 & 0 & 6 & 6 & 7 & 8 & 0 & 0 \\ 0 & 0 & 1 & 6 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In a convolution layer we perform the cross-correlations between the padded bottom and the unaltered kernels like described in section 3.2.1.

stride We already mentioned the stride parameter in section 3.2.1. When s is set to a value greater than 1 we only need to forward the result of the cross-correlation at every s-th position both in the x and y dimension. If we have inputs of size 4×4, kernels of size 2×2, `pad` set to 0 and s set to 2 we cross-correlate the kernel at the bold anchor points in following input only:

$$\begin{bmatrix} \mathbf{7} & 8 & \mathbf{7} & 7 \\ 2 & 2 & 4 & 8 \\ \mathbf{6} & 6 & \mathbf{7} & 8 \\ 1 & 6 & 6 & 2 \end{bmatrix}$$

Note that we use (2.13) for cross-correlating the example and obtain an output of size 2×2 following (2.3):

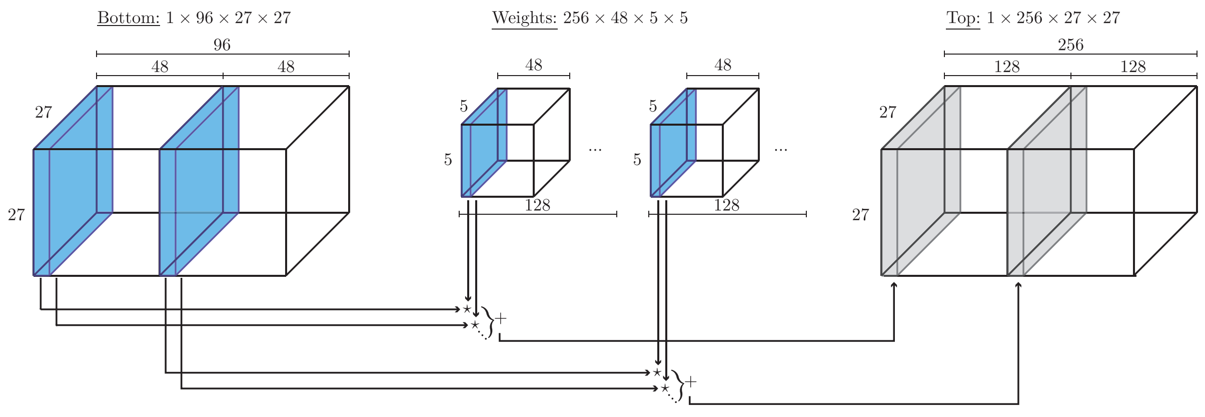
$$\begin{aligned} H_{\text{output}} &= (4 + 2 \cdot 0 - 2)/2 + 1 = 2 \\ W_{\text{output}} &= (4 + 2 \cdot 0 - 2)/2 + 1 = 2 \end{aligned}$$

If we combine `pad` and `stride` we can specify the output size which will be forwarded to the next layer. If we set `pad` to 1 in the our example, we can perform the cross-correlation at more anchor points yielding an output size of 3×3:

$$\begin{bmatrix} \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} & 0 \\ 0 & 7 & 8 & 7 & 7 & 0 \\ \mathbf{0} & 2 & \mathbf{2} & 4 & \mathbf{8} & 0 \\ 0 & 6 & 6 & 7 & 8 & 0 \\ \mathbf{0} & 1 & \mathbf{6} & 6 & \mathbf{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

group We already explained that the `group` parameter splits up the input channels into g groups. This parameter can be used to emulate the setup of multiple GPUs as used in [16]. If set to 2 both the bottom blob and the weights are split up into 2 chunks. We explain differences in the mechanics of the convolution by using the second layer of the reference Caffe network, where `group` $= 2$. In the forward pass the bottom blob is of dimensions $1 \times 96 \times 27 \times 27$ (for a batch size of $N = 1$), whereas we have 256 3D-filters with a size of 5×5 and a depth of 48 each.

In figure 3.3 we demonstrate the convolution with group parameter. We see the bottom blob on the left and its 96 channels divided into two groups. Every group has a size of $1 \times 48 \times 27 \times 27$. In the middle, we see 256 3D-filters divided into two groups of 128 3D-filters with a kernel size of 5×5 and 48 channels each. We perform the convolution straightforward but only between matching groups. We cross-correlate the first half, of the input blob only with weights from the first group. For every of the 128 weights in the first half, we cross-correlate with the first half of the bottom data, sum up over the channels and store the results in the first half of the top blob. This leaves us 128 output features of size 27×27 (Note that we have the same size for the top blob because `pad=2`). We do the same thing for the second group accordingly and, therefore, get a total top size of $1 \times 256 \times 27 \times 27$.



Bottom: $1 \times 96 \times 27 \times 27$
96
48
48
27
27
Weights: $256 \times 48 \times 5 \times 5$
48
5
5
128
...
48
5
5
128
...
Top: $1 \times 256 \times 27 \times 27$
256
128
128
27
27
$\star$
$\star$
+
$\star$
$\star$
+

Figure 3.3.: The convolution as performed in the first layer of the reference Caffe network in the Caffe convolution layer. Cross-correlations are performed for every output feature with the same bottom and forwarded to the top blob.

3.3. New FFT Convolution Layer

For this thesis, we implemented a new convolutional Caffe layer. We generated a subclass of Caffe's `ConvolutionLayer` and called that layer `ConvolutionLayerFFT`. We both implemented CPU and GPU versions written in C++ and CUDA-C respectively. We only needed to alter other code files slightly for Caffe to use the new layer without problems. In this section we give details of our implementation and how we achieve compatibility to Caffe's own convolution layer.

3.3.1. Libraries

For the implementation of our new layer, we used external highly optimized libraries to perform mathematical operations as fast as possible. We use the FFTW [8] library for taking the FFT on the CPU. We actually use Intel's Math Kernel Library (MKL) [5] which supports the FFTW interface and takes advantage of Intel CPUs. On the GPU, we use NVIDIA's CuFFT [7] library which also has an interface similar to the FFTW. We take advantage of a CPU's multi-threading capability by using OpenMP [20] commands. If we link to the MKL library both the FFTW and OpenMP calls are replaced by Intel's own calls within the MKL library. We also use the MKL implementation of BLAS for matrix-related methods on the CPU. On the GPU CuBLAS [6] takes care of the matrix-related functions defined by the BLAS interface.

3.3.2. Preserving Compatibility

We require the new layer to preserve compatibility with Caffe's existing convolution layer. We presented the protocol buffers format in section 3.1.2 and adjusted the definition file in such a way that a convolution layer can easily be replaced by the FFT version. We can create the FFT version of the same layer described in listing 3.1 by using the `engine` parameter of the convolution layer. We extended `engine` by a new option called `FFT`. The definition for the first layer as `ConvolutionLayerFFT` looks like this:

Listing 3.2: CaffeNet's first convolutional layer as FFT version

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    engine: FFT
    num_output: 96
    kernel_size: 11
    stride: 4
  }
}
```

Because we want to compute the same thing but in a different manner we used the existing `engine` parameter since the layer is still a convolution layer but only the computation is done differently. The `layer_factory`'s method for creating a convolution layer parses this argument and decides on the fly if to create a `ConvolutionLayer` or a `ConvolutionLayerFFT` instance.

Because we compute the same thing and want to preserve compatibility within a network (e.g. use `ConvolutionLayer` for layer 1 and `ConvolutionLayerFFT` for the other layers) we have to make sure that our new layer produces the same top blob for a bottom blob as Caffe's layer. We do this by taking the FFT of the input followed by a pointwise multiplication of the inputs and the weights in the frequency domain. At the end, we take the IFFT and we can forward the time domain representation to the next layer. Here any layer can follow expecting a time domain representation of the bottom blob since not implicitly another FFT layer has to follow. This can be helpful because as we have seen in 2.8 not every layer benefits from a convolution in the frequency domain.

3.3.3. Sizes and Memory Allocation

We found out in section 2.5.2 that the FFT only works for certain sample sizes, whereas the Radix-2 FFT is faster than the Radix-3 FFT and so on. We also know that for a convolution in the frequency domain, both the input and the kernel need to have the same size since we can not evaluate a Hadamard product between two matrices of different size. We now show how we can derive a minimal FFT compatible size for H. We can derive the FFT compatible size for W analogously:

$$\begin{aligned} H_{\text{tmp}} &= H + \max(2 \cdot p_h, K_h - 1) \\ H_{\text{FFT}} &= \begin{cases} H_{\text{tmp}} + 16 - (H_{\text{tmp}} \mod 16) & H_{\text{tmp}} \mod 16 > 0 \\ H_{\text{tmp}} & \text{otherwise} \end{cases} \end{aligned} \tag{3.2}$$

We first calculate a temporary size which equals to the minimal size possible needed to calculate the convolution in the frequency domain. If we remember the time domain representation we can instantly see that the bottom height has to be extended by a minimum of $K_h - 1$ to be able to apply the kernel at every last pixel of each bottom's column. Because the convolution in frequency domain is equal to a circular convolution we only need to consider the maximum between `pad` and the kernel size. In table 3.1 we see the optimal FFT sizes for each layer in the reference Caffe network. Note that layer 2–5 have optimal combinations of kernel size, padding and input size. If any of the parameters were larger in these layers we would have to use FFT sizes greater by at least 16.

Table 3.1.: FFT sizes for the reference Caffe network.

	H	W	K_h	K_w	p_h	p_w	H_{FFT}	W_{FFT}
conv1	227	227	11	11	0	0	240	240
conv2	27	27	5	5	2	2	32	32
conv3	13	13	3	3	1	1	16	16
conv4	13	13	3	3	1	1	16	16
conv5	13	13	3	3	1	1	16	16

We used multiples of 16 to satisfy constraints made in the developer documentation of CuFFT [7] to get the best performance out of both the CuFFT and FFTW library. We list these constraints ordered by their importance regarding performance improvements:

1. We use single precision transforms since they require less bandwidth per computation than double precision transforms.

2. We restrict the transform size along all dimensions to be representable as $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. FFT libraries have highly optimized kernels for transforms which have these prime factors. We get the best performances with powers of 2 followed by 3 and so on.

3. We use fewer distinct prime factors along each dimension because the libraries use special computation paths.

4. We have to put the data of a single transform contiguous into the memory. If we want to compute many transforms we have to arrange each dataset contiguous in memory.

5. We have to perform multiple transforms at once since additional optimizations are performed in batch mode by the libraries.

6. We need to ensure that the transform size of the first dimension in real-to-complex transforms is a multiple of 4 because more efficient kernels are used to implement the Hermitian symmetry.

7. We use out-of-place transforms, which have the drawback of consuming nearly double the memory needed for in-place transforms. In section 3.4 we compare those two modus operandi.

We see that a multiple 16 is a perfect size which satisfies constraints 2, 3 and 6. This size promises better performance nearly every time. In [13] non-power-of-two sizes are described to often perform faster than power-of-two problem sizes. If we use the FFTW library on a CPU, sizes of $3600 = 2^4 \cdot 3^2 \cdot 5^2$ and $3840 = 2^8 \cdot 3 \cdot 5$ both perform faster than $4096 = 2^{12}$.

We have to zero-pad the real data before transforming it. We give more details on this procedure in section 3.3.4. Hence, we need to allocate memory for the padded real-valued data and the transformed complex-valued data for out-of-place transforms. Theses sizes evaluate to:

$$\begin{aligned}\texttt{size}^{\mathbb{R}}_{\text{FFT}} &= H_{\text{FFT}} \cdot W_{\text{FFT}} \\ \texttt{size}^{\mathbb{C}}_{\text{FFT}} &= H_{\text{FFT}} \cdot (W_{\text{FFT}}/2 + 1)\end{aligned}$$

Note that we use $\cdot^{\mathbb{R}}_{\text{FFT}}$ for denoting the FFT sizes of real-valued data and $\cdot^{\mathbb{C}}_{\text{FFT}}$ for complex-valued data respectively. In every layer we need to allocate both real and complex arrays for the weights, bottom and top data. We need to allocate memory for every single weight (and channels per weight), input channels and output features for the weight, bottom and top array respectively. The array sizes needed for the weights are

$$\begin{aligned}\texttt{ws}^{\mathbb{R}}_{\text{FFT}} &= N_{\text{output}} \cdot \frac{K}{g} \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}} \\ \texttt{ws}^{\mathbb{C}}_{\text{FFT}} &= N_{\text{output}} \cdot \frac{K}{g} \cdot \texttt{size}^{\mathbb{C}}_{\text{FFT}}.\end{aligned}$$

We need to allocate memory for the padded bottom (remember we get N images in a single batch) and its frequency representation with following sizes:

$$\begin{aligned}\texttt{bs}^{\mathbb{R}}_{\text{FFT}} &= N \cdot K \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}} \\ \texttt{bs}^{\mathbb{C}}_{\text{FFT}} &= N \cdot K \cdot \texttt{size}^{\mathbb{C}}_{\text{FFT}}\end{aligned}$$

At last we need an array to store the complex result of the Hadamard product and its inverse Fourier transform into. We need to allocate this array with following sizes:

$$\begin{aligned}\texttt{ts}_{\text{FFT}} &= N \cdot N_{\text{output}} \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}} \\ \texttt{ts}_{\text{FFT}} &= N \cdot N_{\text{output}} \cdot \texttt{size}^{\mathbb{C}}_{\text{FFT}}\end{aligned}$$

In table 3.2 we see the additional memory each `ConvolutionLayerFFT` needs in contrast to `ConvolutionLayer` when using the reference Caffe network with a batch size of $N = 500$. The table shows the memory needed for single precision values (a single precision value needs 4 bytes of storage and its complex counterpart 8 bytes). If we omit the first layer (which did not promise any speed-up anyway), we need to allocate about 4101 MB of additional memory to store the frequency representations.

3.3.4. Data Preparation for FFT by Zero-Padding

We already allocated the memory to store the zero-padded bottom data and the weights. We have to perform zero-padding in addition to the padding required by the p parameters.

Table 3.2.: Additional memory needed for allocation in each `ConvolutionLayerFFT` of the reference Caffe network with a batch size of 500.

	$\texttt{ws}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{ws}^{\mathbb{C}}_{\text{FFT}}$	$\texttt{bs}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{bs}^{\mathbb{C}}_{\text{FFT}}$	$\texttt{ts}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{ts}^{\mathbb{C}}_{\text{FFT}}$	$\sum$
conv1	63 MB	64 MB	330 MB	332 MB	10547 MB	10635 MB	21971 MB
conv2	48 MB	51 MB	188 MB	199 MB	500 MB	531 MB	1517 MB
conv3	96 MB	108 MB	125 MB	141 MB	188 MB	211 MB	868 MB
conv4	72 MB	81 MB	188 MB	211 MB	188 MB	211 MB	950 MB
conv5	48 MB	54 MB	188 MB	211 MB	125 MB	141 MB	766 MB
$\sum$	327 MB	358 MB	1017 MB	1094 MB	11547 MB	11729 MB	26072 MB

Assume we have an input bottom blob of size $2 \times 3 \times 3 \times 3$ and $p_h = p_w$ set to 2 in this layer. We evaluate $H_{\text{FFT}} = W_{\text{FFT}} = 16$ and, therefore, need to pad each of the 18 single matrices to a matrix of size 16×16. We give an example for zero-padding such a single matrix into an FFT-friendly padded matrix. Note the padding applied within the zero-padding:

$$\begin{bmatrix} 7 & 8 & 7 \\ 7 & 2 & 2 \\ 4 & 8 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&\mathbf{7}&\mathbf{8}&\mathbf{7}&0&0&0&0&0&0&0&0&0&0&0\\
0&0&\mathbf{7}&\mathbf{2}&\mathbf{2}&0&0&0&0&0&0&0&0&0&0&0\\
0&0&\mathbf{4}&\mathbf{8}&\mathbf{6}&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix} \tag{3.3}$$

In the previous example, we showed how to pad a single matrix of a bottom blob. We pad every of the $N \cdot K$ matrices the same way and store the results contiguous in an array holding the padded bottom blob.

We access the unpadded bottom blob by using (3.1) as index:

$$((n \cdot K + k) \cdot H + h) \cdot W + w$$

Following this definition, we can store each value of the unpadded bottom blob at index

$$\begin{aligned}&((n \cdot K + k) \cdot H_{\text{FFT}} + (h + p_h)) \cdot W_{\text{FFT}} + (w + p_w) = \\ &(n \cdot K + k) \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}} + (h + p_h) \cdot W_{\text{FFT}} + (w + p_w)\end{aligned} \tag{3.4}$$

in the padded bottom blob which we initialize with zeros only, whereas $n \in [0; N), k \in [0; K), h \in [0; H), w \in [0; W)$. We see that the first term $(n \cdot K + k) \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}}$ calculates the first pixel of image n and channel k in the padded bottom blob. We add $(h + p_h) \cdot W_{\text{FFT}} + (w + p_w)$ to this index and get the destination index of the pixel h, w in the n-th image with channel k of the unpadded bottom blob.

We can use the same indexing formulas for zero-padding the weight blobs, but we set $p_h = p_w$ to 0. Note that now the bottom blob and the weight have matrices of equal dimensions and we can continue by taking the Fourier transforms of the prepared data.

3.3.5. Taking many FFTs at once

We prepared both the bottom blob and weight blob to have optimal sizes for taking the FFT very effectively. We only need to transform the padded weight blob once since the weights do not change during the forward pass. We can do this the first time we perform a forward pass. Because the weights consist of N_{output} single weights with $\frac{K}{g}$ kernels each, we need to take the 2D-FFT of $N_{\text{output}} \cdot \frac{K}{g}$ time domain representations of the kernels in each layer to get the frequency representation of the weight blob. The bottom blob changes every forward pass since we can pass the network different images every time. In every forward pass, we need to take the 2D-FFT of $N \cdot K$ different matrices.

We discussed performance optimization constraints on parameters for the FFT in the previous section. In the CuFFT library documentation, NVIDIA suggested these to tune Fast Fourier Transforms for best performance. In constraint 5 we stated that we need to perform multiple transforms at once. Both FFTW and CuFFT support this batched mode of the FFT. In both libraries, we have to provide pointers to the source and destination arrays and parameters that exactly define the FFT. First, we need to create a plan where we tell the library on which dimensions to operate. Within the FFTW library we create an FFT plan with the method call `fftwf_plan_many_dft_r2c`. This method creates a plan for many real-to-complex DFTs. We need to provide following parameters:

- **rank** The dimensionality of the transform. In our case, the rank is 2 because we want to perform a 2D-FFT.
- **n** We need to provide the sample size. For the transforms we want to perform, the sample size is an array with values: $\{H_{\text{FFT}}, W_{\text{FFT}}\}$.

- **howmany** The number of transforms to perform in the batch. For the weight transform we need to provide $N_{\text{output}} \cdot \frac{K}{g}$ and $N \cdot K$ for the bottom transform.
- **in** The pointer to the input data. We provide the padded blobs here.
- **inembed** The size of each matrix in the input data. Each padded matrix has a size of $H_{\text{FFT}} \times W_{\text{FFT}}$ so we can provide **n** again.
- **istride** The stride in the input data. Because the padded blob is contiguous in memory, we simply provide 1.
- **idist** The distance in memory between each single matrix to take the FFT for. The distance between each of the transforms is $H_{\text{FFT}} \cdot W_{\text{FFT}}$
- **out** The pointer where to store the frequency domain representation into.
- **onembed** The size of each matrix in the output data. We perform a real-to-complex transform and because of the Hermitian symmetry we need to provide $\{H_{\text{FFT}}, W_{\text{FFT}}/2 + 1\}$ as size for a single output matrix.
- **ostride** The output data is also contiguous in memory and, therefore, we set the output stride to 1 as well.
- **odist** We know the size of each matrix in the output and thus set the distance between two matrices to $H_{\text{FFT}} \cdot (W_{\text{FFT}}/2 + 1)$.
- **flags** We can provide a flag to the planner function where we can choose the method how the library chooses its way to find out the fastest way to compute the FFT. We use `FFTW_ESTIMATE` which uses a simple heuristic that picks a (probably sub-optimal) plan quickly.

After we create the plan, we can simply execute it and the DFT of our input data is written into the output array. As we mentioned before, Intel's MKL provides the same interface as FFTW but uses manufacturer-optimized plans solely designed for Intel CPUs which can yield in better performance. NVIDIA provides an interface very similar to the FFTW interface. We have to provide the same parameters as described above for creating a plan with the exception that we need to pass input and output pointers not until we want to execute the plan.

3.3.6. Hadamard Product in the Frequency Domain

We already know that we have to evaluate the Hadamard Product between the frequency domain representations of the padded bottom and padded weight blob. The underlying idea is quite simple as we only have to take matching values out of each array and perform a complex multiplication. There are some obstacles we want to discuss in this section which may not be immediately obvious.

Evaluating the Hadamard product between two matrices is simple. We just have to find the correct position of those matrices in the memory and store the result at the right position. Remember that we have to reuse the bottom blob for every of the N_{output} output features. In listing 3.3 we see the code for evaluating the Hadamard product on the CPU. The listing shows the naive version we named `pw-simple` in section 2.8.1. In the second line, we calculate the size of a single group. When we set the group parameter to a value greater than 1, a single group's size equals $\frac{1}{g}$ of the total number of weights. We loop over every image in the batch in line 4, over every output feature in line 5 and every pixel inside an image in line 6. Note that variables within the listing are defined slightly different than the previous definitions:

$$\begin{aligned}
\texttt{int batch\_size} &= N\,; \\
\texttt{int N\_output} &= N_{\text{output}}\,; \\
\texttt{int H} &= H_{\text{FFT}}\,; \\
\texttt{int W} &= W_{\text{FFT}}/2+1\,; \\
\texttt{int K} &= \frac{K}{g}\,;
\end{aligned} \tag{3.5}$$

In line 8 we determine in which group we currently operate since $n \in [0; N_{\text{output}})$. Hence, `group_idx` holds a value $\in [0; g)$ indicating the index of the group. In lines 10–11 we set the pointer of the bottom data to the current image in the batch. In lines 13–14 we set the pointer of the result to the corresponding output regarding the current image in the batch. In line 16 we loop over the channels residing in a group. Because the actual number of channels in the bottom blob is $g \cdot \texttt{K}$ (note that `K` is from (3.5) here), we have to calculate the channel index inside the bottom blob in line 17. We calculate the offset to the corresponding weight in line 18. In lines 20–22 we set the pointers of the three arrays to their correct positions and retrieve their values and pointers in lines 24–26 respectively. We reuse the same bottom blob for every output feature. In line 28 we perform a complex multiplication between a single value of the bottom blob and weight blob and add its result into the resulting top array. If we set the third parameter of `fft_cpu_multiply_complex` to `true`, a complex multiplication between the bottom value and the conjugated weight value is performed. We do this because Caffe actually performs a cross-correlation in the convolution layer and we have to multiply with the complex conjugate according to the cross-correlation theorem as described in [2]. Note that we already get rid of the depth dimension here as the Fourier transform is linear. Therefore, it does not matter if we sum up before taking the IFFT or afterward. We reduce the number of IFFTs to take which results in better performance.

Listing 3.3: `pw-simple` way of evaluating the Hadamard product

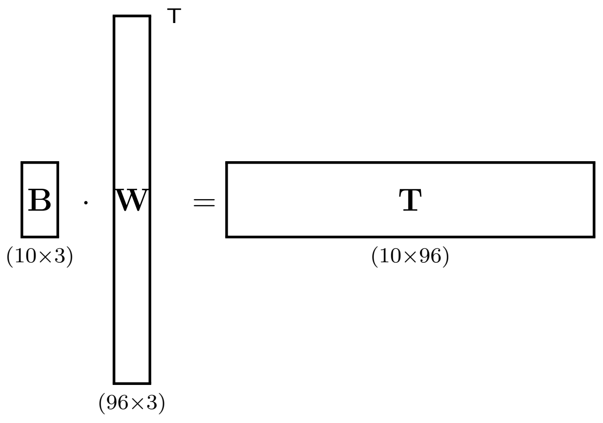
```
1 // Defines the size of a single group.
2 const int weight_group_size = N_output / this->group_;
3
```

```
for (int batch_idx = 0; batch_idx < N; ++batch_idx) {
  for (int n = 0; n < N_output; ++n) {
    for (int hw = 0; hw < H * W; ++hw) {
      // Check which group_idx we are in
      const int group_idx = n / weight_group_size;
      // Calculate the offset in bottom to batch_idx image
      const int bottom_offset = K * this->group_ * H * W * batch_idx;
      std::complex<Dtype> *bottom_data = this_bottom_data + bottom_offset;
      // Calculate the offset in top to batch_idx image
      const int top_offset = N_output * H * W * batch_idx;
      std::complex<Dtype> *top_data = this->ptwise_result + top_offset;

      for (int k = 0; k < K; ++k) {
        const int bottom_k = k + group_idx * K;
        const int weight_offset = (n * K + k);

        const int bottom_idx = bottom_k * H * W + hw;
        const int weight_idx = weight_offset * H * W + hw;
        const int top_idx = n * H * W + hw;

        const std::complex<Dtype> single_bottom = bottom_data[bottom_idx];
        const std::complex<Dtype> single_weight = weight_complex[weight_idx];
        std::complex<Dtype> *single_top = top_data + top_idx;

        *single_top += fft_cpu_multiply_complex(single_bottom, single_weight, true);
      }
    }
  }
}
```

3.3.7. Fast Hadamard Product through Matrix Multiplication

Our implementation of the convolution layer also supports a faster way of computing the Hadamard product as introduced in section 2.7. If we choose this method of computing the Hadamard product we need to permute the dimensions of the frequency domain representations of the bottom and weight blob. We do this right after the frequency domain representations are computed. The permutation transforms the data from a $N \times K \times H \times W$ representation into a $H \times W \times N \times K$ representation. We can easily do this by interchanging the indices in (3.1) accordingly.

We know that in this case $H = H_{\mathrm{FFT}}$ and $W = W_{\mathrm{FFT}}/2 + 1$ for both the bottom blob and weight blob. As we already explored, we now need to perform $H_{\mathrm{FFT}} \cdot (W_{\mathrm{FFT}}/2 + 1)$ matrix multiplications. We see that the inner dimensions of the permuted frequency representations of the bottom blob and weight blob agree in such a way that we are able to perform matrix multiplications. We now use layer 1 of the reference Caffe network and the batch size is 10. We need to perform $240 \cdot 121$ matrix multiplications with the sizes shown:



BLAS libraries also offer methods to compute complex matrix multiplications. We perform these matrix multiplications in a single batch. This way the libraries (we use MKL on the CPU and CuBLAS on the GPU) take care of efficiently distributing the matrix multiplications along threads. We have to prepare three arrays of pointers which point to the correct positions of each single matrix in memory before calling the batched matrix multiplication call. We now give the size of a single bottom matrix, weight matrix, and top matrix respectively:

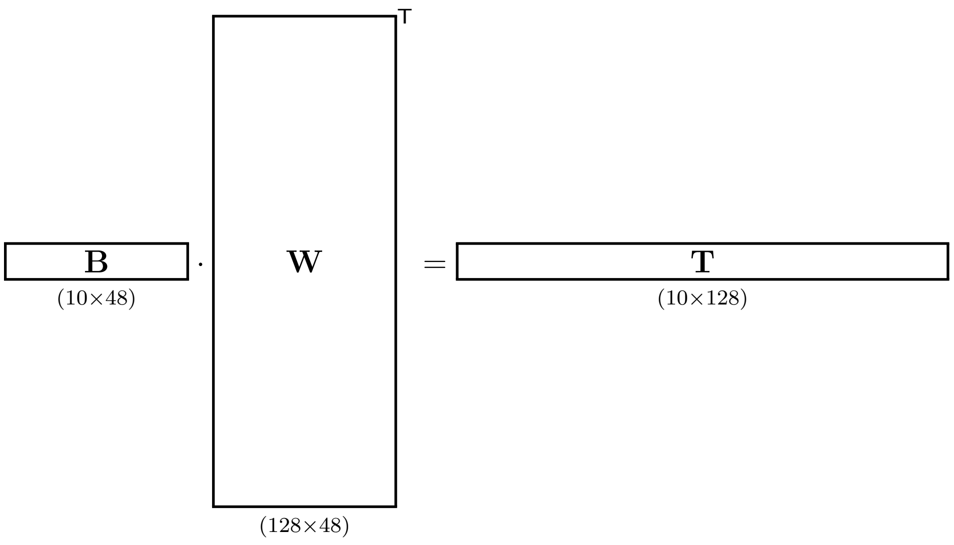
$$\begin{aligned}\texttt{size}_{\text{bottom}} &= N \cdot K \\ \texttt{size}_{\text{weight}} &= N_{\text{output}} \cdot \frac{K}{g} \\ \texttt{size}_{\text{top}} &= N_{\text{output}} \cdot N\end{aligned}$$

We can easily calculate the offsets with respect to the beginning of the permuted blobs to receive the pointers for corresponding matrices:

$$\begin{aligned}\texttt{ptr}^{h,w}_{\text{bottom}} &= \texttt{ptr}^{0,0}_{\text{bottom}} + (h \cdot W + w) \cdot \texttt{size}_{\text{bottom}} \\ \texttt{ptr}^{h,w}_{\text{weight}} &= \texttt{ptr}^{0,0}_{\text{weight}} + (h \cdot W + w) \cdot \texttt{size}_{\text{weight}} \\ \texttt{ptr}^{h,w}_{\text{top}} &= \texttt{ptr}^{0,0}_{\text{top}} + (h \cdot W + w) \cdot \texttt{size}_{\text{top}}\end{aligned} \tag{3.6}$$

Note that $h \in [0; H), w \in [0; W)$ and $H = H_{\text{FFT}}$ and $W = W_{\text{FFT}}/2 + 1$ like we explained before. $\texttt{ptr}^{0,0}_{\text{bottom}}$, $\texttt{ptr}^{0,0}_{\text{weight}}$, $\texttt{ptr}^{0,0}_{\text{top}}$ are pointers to the beginning of the permuted bottom, weight or top blob respectively. While these definitions would suffice for the case were $g = 1$ we have to perform even more but smaller matrix multiplications in the case were $g > 1$. We need to do this for every group. Thus, we need to perform 2 times the matrix multiplications if $g = 2$. In layer 2 of the reference Caffe network we need to perform

$32 \cdot 17 \cdot 2$ matrix multiplications as shown below:



We need to add offsets to the pointer locations calculated in (3.6) for each of the groups. The offset of a single group can be obtained this way:

$$\begin{aligned} \texttt{offset}_{\text{bottom}} &= \frac{\texttt{size}_{\text{bottom}}}{N \cdot g} \\ \texttt{offset}_{\text{weight}} &= \frac{\texttt{size}_{\text{weight}}}{g} \\ \texttt{offset}_{\text{top}} &= \frac{\texttt{size}_{\text{top}}}{N \cdot g} \end{aligned} \tag{3.7}$$

We can now determine the pointers of all matrices belonging together by adding the index of a group ($\hat{g} \in [0; g)$) to the equations introduced in (3.6):

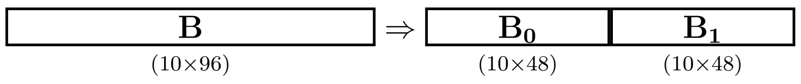
$$\begin{aligned} \texttt{ptr}^{h,w,\hat{g}}_{\text{bottom}} &= \texttt{ptr}^{h,w}_{\text{bottom}} + \hat{g} \cdot \texttt{offset}_{\text{bottom}} \\ \texttt{ptr}^{h,w,\hat{g}}_{\text{weight}} &= \texttt{ptr}^{h,w}_{\text{weight}} + \hat{g} \cdot \texttt{offset}_{\text{weight}} \\ \texttt{ptr}^{h,w,\hat{g}}_{\text{top}} &= \texttt{ptr}^{h,w}_{\text{top}} + \hat{g} \cdot \texttt{offset}_{\text{top}} \end{aligned} \tag{3.8}$$

We can easily fill the three pointer arrays by storing all corresponding pointers which we calculate by iterating over all H, W, g and using (3.8) to find the right offsets in memory.

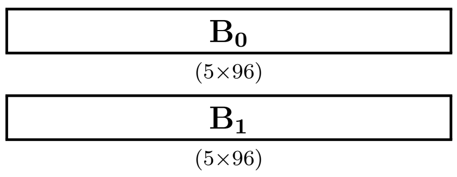
The BLAS call for performing a batched complex matrix multiplication with single precision is called `cblas_cgemm_batch` and is only available in the Intel MKL and CuBLAS libraries. We have to provide three arrays of pointers, which point to a memory location where the corresponding matrix is located. We also need to tell the method how to interpret the source matrices before performing the actual matrix multiplication. For the bottom matrices, we provide the parameter `CblasNoTrans` which says not to transpose the matrices. We need to perform multiplications with the complex conjugate as discovered in 3.3.6 and also need to transpose those matrices like we visualized before.

Thus, we provide `CblasConjTrans` for the weights matrices. Of course we also need to provide the matrix sizes $\dot{M}, \dot{N}$ and $\dot{K}$ ($\dot{M} = N, \dot{N} = \frac{N_{\text{output}}}{g}, \dot{K} = \frac{K}{g}$).

Even though we took care of the group parameter when we calculated the offsets within the blobs, the matrices are in a wrong order if the batch size parameter is set to a value greater than 1. We use the second layer of the reference Caffe network to demonstrate the data layout with group parameter set to $g = 2$ and the batch size set to $N = 10$. After permuting the bottom blob the dimensions of each bottom matrix are 10×96, but actually the bottom blob has to be divided into 2 groups ($\mathbf{B_0}$ and $\mathbf{B_1}$):



If we pass a pointer to each matrix **B** in `cblas_cgemm_batch` and also tell the method that the size of a bottom matrix is 10×48 not $\mathbf{B_0}$ will be used, but the first 480 values within **B** will be interpreted as $\mathbf{B_0}$. **B** is stored in row-major order so `cblas_cgemm_batch` would actually treat **B** for the first and second group like this:



We can provide a parameter that specifies the leading dimension for every of the matrices involved in the matrix multiplication. If we set the leading dimension to $2 \cdot 48$ for the bottom matrices the group parameter will be handled correctly. Because the order in the top blob conforms to the order in the top blob we have to adjust its leading dimension accordingly. We set the leading dimension of the three matrices to the following values:

$$\begin{aligned} \texttt{ld}_{\text{bottom}} &= \dot{K} \cdot g = K \\ \texttt{ld}_{\text{weight}} &= \dot{K} = \frac{K}{g} \\ \texttt{ld}_{\text{top}} &= \dot{N} \cdot g = N_{\text{output}} \end{aligned} \tag{3.9}$$

Altogether we now perform the matrix multiplications needed to get a permuted top blob. If we reverse the permutation, we get the equal result as calculated with `pw-simple` in the section before. We named the fast method of performing the Hadamard product `pw-cgemm` in section 2.8.2 when we estimated possible speed-ups. As with `pw-simple`, we now need to take the IFFT and normalize the real-valued result.

3.3.8. IFFT and Normalization

In the previous sections, we computed the cross-correlation in the frequency domain, but the following layers expect real-valued inputs. These non-linearity layers require time domain representations of the data. Therefore, we need to take the IFFT of the complex-valued Hadamard product and write it into the top blob, which is forwarded to the next layers. We need to perform multiple inverse transforms here. For every image in a batch and output feature, we take the IFFT. Hence we can use a batched version to process $N \cdot N_{\text{output}}$ transforms. Taking the IFFT with libraries that support FFTW like interfaces is very simple. In 3.3.5 we introduced `fftwf_plan_many_dft_r2c` for taking many single precision real-to-complex DFTs. We take the IFFT by simply using the planner for complex-to-real DFTs, which creates an IFFT plan. `fftwf_plan_many_dft_c2r` takes the same parameters as the real-to-complex planner method. We only need to make sure to inverse some of the input and output parameters in contrast to the real-to-complex planner method:

- **in** We have to provide the pointer where the Hadamard product's result is stored. In the case of listing 3.3 this would be `this->ptwise_result`.
- **inembed** The input matrices' sizes are $\{H_{\text{FFT}}, W_{\text{FFT}}/2 + 1\}$ now.
- **istride** The stride of the Hadamard product's result is also 1.
- **idist** The distance in memory between two single complex-valued matrices results from **inembed**: $H_{\text{FFT}} \cdot (W_{\text{FFT}}/2 + 1)$
- **out** The pointer to an array where the real-valued result of the cross-correlation should be stored.
- **onembed** The size of each real-valued matrix is the same as for a bottom blob or a weight blob: $\{H_{\text{FFT}}, W_{\text{FFT}}\}$
- **ostride** We want the result to be contiguous in memory so we set ostride to 1 here.
- **odist** Follows from **onembed**: $H_{\text{FFT}} \cdot W_{\text{FFT}}$.

We now have a real-valued blob containing values which we want to write into the top blob, but we have to normalize those values and pick the right ones to write into the top blob. Each matrix has a size of $H_{\text{FFT}} \times W_{\text{FFT}}$ but we need to forward matrices of size $H_{\text{output}} \times W_{\text{output}}$. If we look at the first layer, we need to reduce the size from 240×240 to 55×55. Because we already applied p_h and p_w when zero-padding, the bottom blob valid cross-correlation results are located the top left corner of each matrix in the real-valued blob. Note that we did not apply the stride parameter of the convolution layer yet, which is not possible in the frequency domain. We performed a cross-correlation at every position and need to write every s-th pixel of each dimension into the top blob. We also need to normalize every value by $\frac{1}{H_{\text{FFT}} \cdot W_{\text{FFT}}}$ like described in (2.10) before we write into the top blob.

In listing 3.4 we show how the normalization is handled in our implementation. We first calculate the normalization term $\frac{1}{H_{\text{FFT}} \cdot W_{\text{FFT}}}$ in line 2 and then loop over `N`, `K`, `H`, `W` which are set to N, N_{output}, H_{output}, W_{output} respectively. For every single matrix needed to store in the top blob, we calculate the offset within our real-valued result in line 5. In lines 13–14 we calculate the index in the real-valued blob based on s and p. p is set to 0 in this particular case, because valid results are stored in the top-left corner of the real-valued matrices. (We need p for other cases we explain in section 3.3.10.) In lines 16–17 we calculate corresponding indexes in the real-valued blob and top blob. We normalize the picked value in line 20 and write it into the top blob in line 24. (The parameter `add_to_result` is set to `false` and will be discussed in section 3.3.10.)

Listing 3.4: Applying stride and normalization to cross-correlation result

```
 1 Dtype ifft_normalize_factor = 1. / (fft_width * fft_height);
 2
 3 for (int n = 0; n < N; ++n) {
 4   for (int k = 0; k < K; ++k) {
 5     const int offset_ifft_real = (n * K + k) * (fft_width * fft_height);
 6
 7     for (int h = 0; h < H; ++h)
 8     {
 9       for (int w = 0; w < W; ++w)
10       {
11         // The first valid result in the real conv array is at pad_h, pad_w.
12         // We step by the stride to pick the values of interest.
13         int h_idx = pad_h + h * stride_h;
14         int w_idx = pad_w + w * stride_w;
15
16         const int top_idx = ((n * K + k) * H + h) * W + w;
17         const int ifft_real_idx = offset_ifft_real + h_idx * fft_width + w_idx;
18
19         // Normalize result and write it into top blob
20         Dtype tmp_result = iffted_result[ifft_real_idx] * ifft_normalize_factor;
21         if (add_to_result) {
22           top[top_idx] += tmp_result;
23         } else {
24           top[top_idx] = tmp_result;
25         }
26       }
27     }
28   }
29 }
```

3.3.9. Forward Pass

We introduced every necessary part to perform a forward pass of a convolution layer in the frequency domain. The `ConvolutionLayerFFT` now produces the exact same result like Caffe's own `ConvolutionLayer`. We assemble the parts of the previous sections and give an overview over a complete forward pass, which we visualize in figure 3.4. We only need to set up the FFT convolution layer (`fft_set_up()`) in the first forward pass. Here we initialize sizes and allocate the memory needed in the layer like in section 3.3.3. We need to pad the real-valued weight blob (see section 3.3.4) before taking the real-to-complex

FFT. If we choose to use the fast Hadamard product (`pw-cgemm`) we need to permute the frequency representation of the weight blob before we can continue. Otherwise, we do not need to alter the frequency representation of the weights.

In every single pass, we need to transform the bottom blob, which we do in `fft_bottom()`. Similar to the weight transformation we pad the bottom blob (note that we apply p_h and p_w here like in (3.3)) to be able to take the real-to-complex FFT afterwards. If `pw-cgemm` is enabled, we need to permute the frequency representation of the bottom blob also.

Having computed the frequency representations for both the bottom blob and weight blob, we need to evaluate the Hadamard product now. We can either use the simple way (`pw_simple()`) explained in section 3.3.6 or the fast version (`pw_cgemm()`) described in section 3.3.7. Of course we need to undo the permutation if we use `pw_cgemm()`. At the end, we only need to take the IFFT and normalize the real-valued result as described in section 3.3.8. Note the re-usability of methods like `pad_real_blob()` and `permute_4d()` which we will also need in the backward pass.

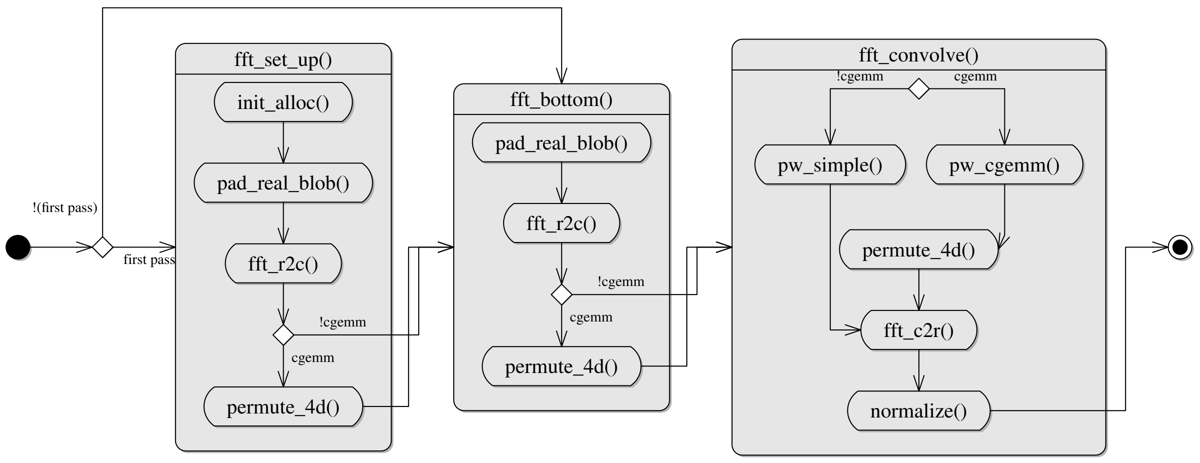


Figure 3.4.: A simplified activity diagram of the forward pass as performed in `ConvolutionLayerFFT`.

3.3.10. Backward Pass

In section 2.9 we derived the backpropagation algorithm for a convolution layer. We concluded that a backward pass in Caffe's convolution layer can be expressed through a cross-correlation and a convolution.

Weight update step At first we need to update weight values like we derived in (2.15). In Caffe terminology this formula is equal to following cross-correlation:

$$\texttt{weight\_diff} = \texttt{bottom} \star \texttt{top\_diff}$$

We get handed the `top_diff` in the backward pass as this is the error in the current layer which we know already. Like in the forward pass we need to zero-pad the bottom blob and take its FFT. Obviously, we need to process `top_diff` the same way before we can compute the cross-correlation in the frequency domain. Because the size of `top_diff`, which has the same size as the top blob, is often smaller than the bottom blob we need to correctly upscale this data at first. Thereto we alter the padding algorithm presented in section 3.3.4 by extending (3.4) to support a stride parameter s as well:

$$(n \cdot K + k) \cdot \texttt{size}^{\mathbb{R}}_{\text{FFT}} + (h \cdot s + p_h) \cdot W_{\text{FFT}} + (w \cdot s + p_w) \tag{3.10}$$

If we use $s = 3$ for our example demonstrated in (3.3) we get a slightly different result:

$$\begin{bmatrix} 7 & 8 & 7 \\ 7 & 2 & 2 \\ 4 & 8 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{7} & 0 & 0 & \mathbf{8} & 0 & 0 & \mathbf{7} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{7} & 0 & 0 & \mathbf{2} & 0 & 0 & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{4} & 0 & 0 & \mathbf{8} & 0 & 0 & \mathbf{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix} \tag{3.11}$$

We use (3.10) to zero-pad `top_diff` with $p_h = p_w$ set to 0 and s set to the layer's stride parameter. Every other parameter is set as explained in section 3.3.4. Now we can easily evaluate the Hadamard product, take the IFFT and normalize the result yielding `weight_diff`. For normalizing we use $s = 1$ and $p_h = p_w = 0$. Caffe automatically updates the real-valued weights accordingly. We reuse the memory allocated in section 3.3.3 since the sizes match but we only use a different order of evaluation. Afterwards we normalize with `stride_h` = `stride_w` = 1 and `pad_h` = `pad_w` = 0 with the algorithm presented in listing 3.4. Because we accumulate the `weight_diff` onto the old `weight_diff`, we set `add_to_result` to `true` here.

Propagating error to bottom layer We need to propagate the error to the previous layer. We express (2.16) like this in Caffe terminology:

$$\texttt{bottom\_diff} = \texttt{top\_diff} * \texttt{weight}$$

Note that this formula now actually requires a convolution ($*$) to be computed instead of a cross-correlation($\star$). Therefore, we do not multiply by the complex conjugate. We need to pad the `top_diff` like before and evaluate Hadamard product with the updated weights. Hence, we have to update the frequency representation of the weights before. Again we use the same memory we allocated in section 3.3.3. For normalization we set `stride_h` = `stride_w` = 1 and `pad_h` = p_h, `pad_w` = p_w. We need to set `add_to_result` to `false`.

Adjusting parameters of fast Hadamard product We introduced parameters used by `pw-cgemm` to properly multiply corresponding matrices with each other. In table 3.4.1 we give an overview over different values needed to perform the cross-correlations and convolution respectively. We named the single cross-correlation in the forward pass **FW**. In the backward pass we called the weight update cross-correlation **W** while we denoted the backpropagation convolution by **BW**. The kind of cross-correlation/convolution is denoted in the first column of the table. Note the $\cdot^{\mathsf{H}}$ on top of some matrices which indicates the conjugate transpose, while $\cdot^{\mathsf{T}}$ denotes a simple transpose. In the second column we visualize the matrix-multiplications which will be performed $H_{\mathrm{FFT}} \cdot (W_{\mathrm{FFT}}/2+1)$ times. All sizes are given for the second layer in the reference Caffe network therefore yielding 2 matrix multiplications (one for each group) per cross-correlation/convolution. Every matrix multiplication in the second column is of the form $\mathbf{a} \cdot \mathbf{b} = \mathbf{c}$. We used **B**, **W** and **T** for the bottom blob, weight blob and top blob respectively. A prefixed ∇ operator indicates the corresponding `_diff` blob. In the third column the actual resulting matrix out of the two matrix-multiplications in showed. We list the corresponding sizes as introduced for the forward pass in section 3.3.7 in columns 4–9. The indices $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are chosen to match the order of matrix multiplication in column 2 ($\mathbf{a} \cdot \mathbf{b} = \mathbf{c}$).

3.4. Further Optimizations

In this section, we describe further optimizations adopted to increase the performance of the frequency domain convolution in our implementation of the convolution layer. We implemented both CPU and GPU versions of these optimizations.

3.4.1. Parallelizing Workload

We took advantage of the performance boosts possible by distributing workload onto multiple CPU cores or CUDA cores on NVIDIA GPUs. We enabled OpenMP [20] on the CPU, thereupon, MKL automatically parallelizes its FFT implementation. We furthermore enabled all our loops to distribute the workload among threads.

	CGEMM group 0	Result	$\mathtt{size}_{\mathbf{a}}$	$\mathtt{size}_{\mathbf{b}}$	$\mathtt{size}_{\mathbf{c}}$	$\mathtt{ld}_{\mathbf{a}}$	$\mathtt{ld}_{\mathbf{b}}$	$\mathtt{ld}_{\mathbf{c}}$
FW	$\underset{(10\times48)}{\boxed{\mathbf{B}}} \cdot \underset{(128\times48)}{\boxed{\mathbf{W}}}^{\mathsf{H}} = \underset{(10\times128)}{\boxed{\mathbf{T}}}$	$\underset{(10\times256)}{\boxed{\mathbf{T}}}$	$N \cdot K$	$N_{\text{output}} \cdot \frac{K}{g}$	$N_{\text{output}} \cdot N$	K	$\frac{K}{g}$	N_{output}
	CGEMM group 1		$\mathtt{offset}_{\mathbf{a}}$	$\mathtt{offset}_{\mathbf{b}}$	$\mathtt{offset}_{\mathbf{c}}$			
	$\underset{(10\times48)}{\boxed{\mathbf{B}}} \cdot \underset{(128\times48)}{\boxed{\mathbf{W}}}^{\mathsf{H}} = \underset{(10\times128)}{\boxed{\mathbf{T}}}$		$\frac{\mathtt{size}_{\mathbf{a}}}{N\cdot g}$	$\frac{\mathtt{size}_{\mathbf{b}}}{g}$	$\frac{\mathtt{size}_{\mathbf{c}}}{N\cdot g}$			
	CGEMM group 0	Result	$\mathtt{size}_{\mathbf{a}}$	$\mathtt{size}_{\mathbf{b}}$	$\mathtt{size}_{\mathbf{c}}$	$\mathtt{ld}_{\mathbf{a}}$	$\mathtt{ld}_{\mathbf{b}}$	$\mathtt{ld}_{\mathbf{c}}$
W	$\underset{(10\times128)}{\boxed{\nabla\mathbf{T}}}^{\mathsf{H}} \cdot \underset{(10\times48)}{\boxed{\mathbf{B}}} = \underset{(128\times48)}{\boxed{\nabla\mathbf{W}}}$	$\underset{(256\times48)}{\boxed{\nabla\mathbf{W}}}$	$N_{\text{output}} \cdot N$	$N \cdot K$	$N_{\text{output}} \cdot \frac{K}{g}$	N_{output}	K	$\frac{K}{g}$
	CGEMM group 1		$\mathtt{offset}_{\mathbf{a}}$	$\mathtt{offset}_{\mathbf{b}}$	$\mathtt{offset}_{\mathbf{c}}$			
	$\underset{(10\times128)}{\boxed{\nabla\mathbf{T}}}^{\mathsf{H}} \cdot \underset{(10\times48)}{\boxed{\mathbf{B}}} = \underset{(128\times48)}{\boxed{\nabla\mathbf{W}}}$		$\frac{\mathtt{size}_{\mathbf{a}}}{N\cdot g}$	$\frac{\mathtt{size}_{\mathbf{b}}}{N\cdot g}$	$\frac{\mathtt{size}_{\mathbf{c}}}{g}$			
	CGEMM group 0	Result	$\mathtt{size}_{\mathbf{a}}$	$\mathtt{size}_{\mathbf{b}}$	$\mathtt{size}_{\mathbf{c}}$	$\mathtt{ld}_{\mathbf{a}}$	$\mathtt{ld}_{\mathbf{b}}$	$\mathtt{ld}_{\mathbf{c}}$
BW	$\underset{(10\times128)}{\boxed{\mathbf{T}}} \cdot \underset{(128\times48)}{\boxed{\mathbf{W}}} = \underset{(10\times48)}{\boxed{\nabla\mathbf{B}}}$	$\underset{(10\times96)}{\boxed{\nabla\mathbf{B}}}$	$N_{\text{output}} \cdot N$	$N_{\text{output}} \cdot \frac{K}{g}$	$N \cdot K$	K	$\frac{K}{g}$	N_{output}
	CGEMM group 1		$\mathtt{offset}_{\mathbf{a}}$	$\mathtt{offset}_{\mathbf{b}}$	$\mathtt{offset}_{\mathbf{c}}$			
	$\underset{(10\times128)}{\boxed{\mathbf{T}}} \cdot \underset{(128\times48)}{\boxed{\mathbf{W}}} = \underset{(10\times48)}{\boxed{\nabla\mathbf{B}}}$		$\frac{\mathtt{size}_{\mathbf{a}}}{N\cdot g}$	$\frac{\mathtt{size}_{\mathbf{b}}}{g}$	$\frac{\mathtt{size}_{\mathbf{c}}}{N\cdot g}$			

Table 3.3.: Parameters for the three `cgemm` calls needed for the forward (FW) and backward pass (W and BW).

Our algorithm for zero-padding the data introduced in section 3.3.4 loops over 4 dimensions, which allowed us to use an OpenMP notation for condensing the loops and redistributing the workload among different threads. Every possible thread accesses distinct locations in memory, hence, we did not need to synchronize concurrent memory accesses.

In listing 3.3 we introduced the simple algorithm for evaluating the Hadamard product. We could also easily parallelize the three loops in lines 4–6 with OpenMP. The last loop in line 16 is looping over the channels and summing the results up in line 28. Because we access the same memory location for each of the K channels in this line we cannot parallelize this last loop, otherwise resulting in indeterminate results. For `pw-cgemm`, we used the batched matrix multiplication call within Intel's MKL library, which automatically distributes single matrix multiplications among threads yielding a high performance. We also parallelized the normalization algorithm introduced in listing 3.4 by using OpenMP commands. Here we parallelized over the first two loops.

We also implemented all parallelizations presented here for the GPU. GPUs support heavy parallelizing without the use of OpenMP and are designed to fulfill such tasks very effectively.

3.4.2. Fast 4D Transpose

We already proposed a fast way for evaluating the Hadamard product in sections 2.7 and 3.3.7. In the same paper [25] we adapted this idea from, a fast way for permuting the data beforehand is proposed. The certain transposition from a $N \times K \times H \times W$ layout to a $H \times W \times N \times K$ layout preserves the order and only interchanges the parts separated after the second dimension ($N \times K \rightleftarrows H \times W$). In section 3.3.7 we introduced the simple idea to simply change the indexing dimension in (3.1). Because of non-consecutive random memory accesses this idea slows `pw-cgemm` down so much, it even performs slower than `pw-simple`. In both the MKL and CuBLAS libraries methods for transposing 2D matrices exist. We can take advantage of those tuned methods by expressing our 4D transpose as a 2D matrix transpose. Assume we have a blob with a size of $3 \times 2 \times 2 \times 2$:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \Bigg| \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \Bigg| \begin{bmatrix} 17 & 18 \\ 19 & 20 \end{bmatrix} \begin{bmatrix} 21 & 22 \\ 23 & 24 \end{bmatrix}$$

If we transpose this to be of a $2 \times 2 \times 3 \times 2$ layout the result looks like this:

$$\begin{bmatrix} 1 & 5 \\ 9 & 13 \\ 17 & 21 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 10 & 14 \\ 18 & 22 \end{bmatrix} \Bigg| \begin{bmatrix} 3 & 7 \\ 11 & 15 \\ 19 & 23 \end{bmatrix} \begin{bmatrix} 4 & 8 \\ 12 & 16 \\ 20 & 24 \end{bmatrix}$$

We can interpret the row-major representation of the source blob as a 2D-matrix of size $H \cdot W \times N \cdot K = 6 \times 4$ and 2D transpose this matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \\ 4 & 8 & 12 & 16 & 20 & 24 \end{bmatrix}$$

If we reinterpret the row-major memory of the transposed matrix as an 4D-array with a size of $H \times W \times N \times K$ we get the desired result in a fast and highly optimized way. We can easily 2D transpose our blobs with methods available in Intel's MKL (not in other CPU BLAS libraries) and in the NVIDIA CuBLAS library.

3.4.3. In-place FFT

We also introduce an option to specify the kind of FFT to take within the protocol buffer definition file. We name this parameter `fft_inplace`, which is set to false by default. In section 3.3.3 we allocated both space for the real-valued blob and its complex-valued frequency representations. To save memory, we can only allocate the complex-valued arrays and perform an in-place FFT instead of an out-of-place FFT. We forfeit a little performance while conserving valuable memory on the GPU. We prioritized an out-of-place FFT last within the list of constraints proposed in section 3.3.3, which allows us to use less memory while a drop in performance is hardly noticeable.

In table 3.4 we depict the memory savings for layers 2–5 with a batch size of $N = 500$ in the reference Caffe network. We highlighted the sizes used for allocating real-valued memory in red. We do not need to allocate these in an in-place FFT mode. In the last two columns, we see both memory needed for the real-valued and complex-valued representations respectively. We can actually save around 1952 MB of memory by using the in-place FFT, which is quite a lot even on modern GPUs. Thus, we only need to allocate additional 2149 MB of memory instead of 4101 MB.

Table 3.4.: Memory savings in `ConvolutionLayerFFT` through using an in-place FFT instead of an out-of-place FFT.

	$\texttt{ws}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{ws}^{\mathbb{C}}_{\text{FFT}}$	$\texttt{bs}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{bs}^{\mathbb{C}}_{\text{FFT}}$	$\texttt{ts}^{\mathbb{R}}_{\text{FFT}}$	$\texttt{ts}^{\mathbb{C}}_{\text{FFT}}$	$\sum^{\mathbb{R}}$	$\sum^{\mathbb{C}}$
conv2	48 MB	51 MB	188 MB	199 MB	500 MB	531 MB	736 MB	781 MB
conv3	96 MB	108 MB	125 MB	141 MB	188 MB	211 MB	408 MB	460 MB
conv4	72 MB	81 MB	188 MB	211 MB	188 MB	211 MB	447 MB	503 MB
conv5	48 MB	54 MB	188 MB	211 MB	125 MB	141 MB	360 MB	406 MB
$\sum$	264 MB	294 MB	688 MB	762 MB	1000 MB	1094 MB	1952 MB	2149 MB

For the in-place FFT to work, we need to store the real-valued data in the complex array with correct in-memory strides that correspond to those used for the complex array. The stride between two single matrices is $H \cdot W$ in the real valued array whereas we have to use a stride of $H \cdot (W/2+1) \cdot 2$ when we store real-valued data in a complex array. Note that we need to multiply by 2, because a complex value needs twice the storage than a real value.

3.5. GPU Implementation

Most implementation details discussed in the previous sections focused on the CPU implementation. We also implemented a GPU version of the `ConvolutionLayerFFT` which we optimized to take advantage of the architecture of NVIDIA GPUs. We translated all loops parallelized with OpenMP in the CPU implementation into CUDA kernels. A CUDA kernel [15] is an extremely lightweight function running on the GPU with little creation overhead. CUDA is defined by the ability to switch between those kernels instantly. In contrast to a CPU, CUDA uses thousands of threads which each executes the same kernel. Each thread has an ID which it uses to calculate indices and memory addresses. We implemented the innermost parts of our loops as kernels. CUDA also comes with tuned libraries such as CuFFT and CuBLAS, which support fast Fourier transforms and a BLAS interface respectively. We used CuFFT to take the Fourier and inverse Fourier transforms while we used CuBLAS for matrix related operations such as matrix transposition and matrix multiplication.

4. Evaluation

In section 2.8 we calculated theoretical speed-ups our method of computing the convolution can achieve within a convolution layer. In this chapter we test our layer with Caffe's built-in benchmark mode. We test CPU and GPU implementations and compare them with Caffe's own convolution layer. For each implementation we compare the performance by altering parameters such as the batch size, the method of evaluating the Hadamard product or using an in-place FFT instead of an out-of-place FFT. We first test with a real-world example of a DCNN - the reference Caffe network, which is based on the ImageNet introduced by Krizhevsky et al. [16]. Moreover we compare the training of the complete reference Caffe network with our method in contrast to a conventional training. Afterward, we spend some thoughts on how to design convolution layer's sizes in a way to achieve even better performances. Finally, we assure our implementation yields the exact same results as Caffe's convolution layer.

4.1. Test Setup

We measured all performance tests on workstation running Ubuntu 14.04.3 LTS Server. The CPU we used is an Intel® Xeon® Processor E5-2680 v2 with 10 cores, 20 threads and 25 MB of cache. We had 64 GB of usable main memory. All GPU performance tests were conducted on a NVIDIA® GeForce GTX TITAN X with 12288 MB of memory and 3072 CUDA cores.

We time forward and backward passes of the reference Caffe network by using the validation set provided by the ILSVRC challenge of 2012 [22] containing 50,000 images. We train our network in section 4.3 using the ILSVRC 2012 training set containing 1.2 million images.

We changed the protocol buffer definitions of the reference Caffe network in a test network file, which contains uses ILSVRC 2012 validation set as data source. We altered the batch size for some of our tests. We also changed the protocol buffer file in such a way, that we both use the FFT convolution layer and the standard convolution layer.

4.2. Performance on the Reference Caffe Network

We now compare our implementation of the convolution layer versus Caffe's own convolution layer on the reference Caffe network (see section 2.3 for the architecture). First, we compare the performance on the GPU, because this is the preferred mode for computing DCNNs in Caffe. We test both the forward pass and the weight update step of the backward pass with Caffe's `time` mode, which times processing time in each layer and averages automatically over the number of iterations. For every test we run 200 iterations and use the average time measured. Because the first layer of the reference Caffe network does not promise any possible speed-ups, but even decreases the network's performance as we found out in section 2.8, we only use a `ConvolutionLayerFFT` in convolution layers 2–5. In the first layer we always use a conventional `ConvolutionLayer`. We name the Caffe convolution, the FFT convolution using the simple Hadamard product and the FFT convolution using the fast Hadamard product `caffe-conv`, `fft-conv` and `fft-cgemm-conv` respectively. These naming conventions conform to those introduced in section 2.8.

4.2.1. GPU Performance

4.2.1.1. Forward Pass

For the timings of the forward pass we use a fixed batch size of $N = 400$ for comparing the Caffe convolution with the FFT convolutions `fft-conv` and `fft-cgemm-conv`. Later we compare the performance with a variable batch size.

caffe-conv vs. fft-conv In table 4.1 we show times in milliseconds per layer. We can confirm our theoretical assumptions depicted in figure 2.8 when we compare `caffe-conv` and `fft-conv`. Only convolution layer 2 shows a small speed-up ($\approx 1.42\times$) while convolution layers 3–5 perform worse. `conv1` uses `caffe-conv` in both cases, hence, the performance is nearly identical. `pass` denotes a complete forward pass including other layers of the network not shown in the table. We visualized the results in figure 4.1.

caffe-conv vs. out-of-place fft-cgemm-conv We compare the traditional Caffe convolution to the out-of-place `fft-cgemm-conv`. Here our assumptions visualized in figure 2.10 also prove to be accurate. We can observe a heavy improvement in performance for layers 2–5 in table 4.2. Note that `conv1`'s method is `caffe-conv` in both cases again and the times are basically the same. The total speed-up for a forward pass is $2.18\times$ with our method. We visualize this speed-up in figure 4.2.

Table 4.1.: GPU Forward times (in ms) of `caffe-conv` and `fft-conv` in comparison.

Layer	`fft-conv` ($N = 400$)	`caffe-conv` ($N = 400$)
conv1	103.02	100.86
conv2	109.97	155.78
conv3	187.00	103.43
conv4	147.42	114.84
conv5	101.20	93.20
pass	693.74	613.21

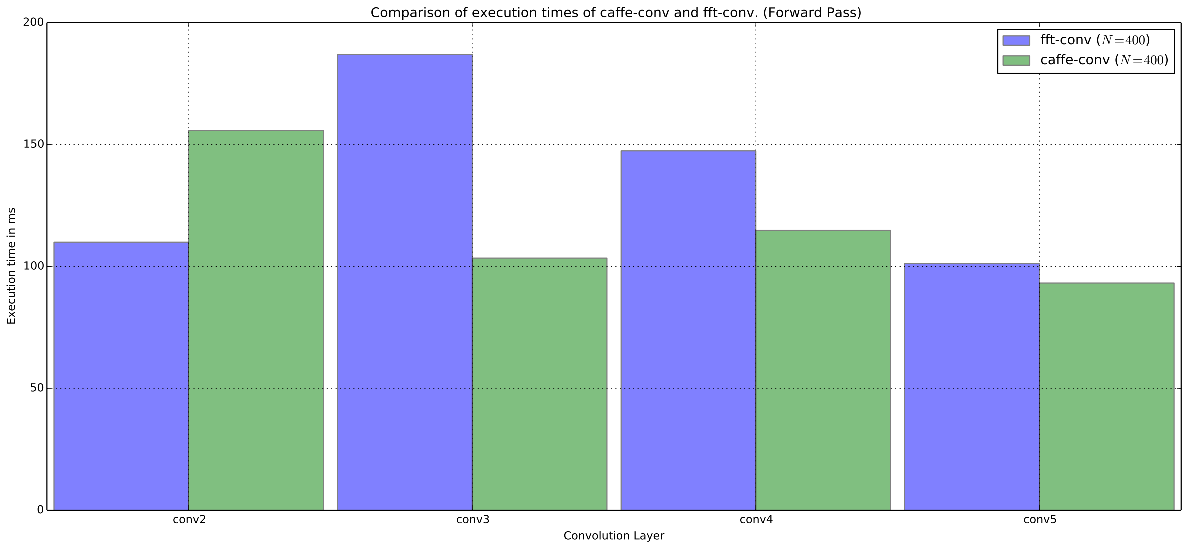


Figure 4.1.: Comparison of execution times of `caffe-conv` and `fft-conv` in the reference Caffe network. (GPU Forward Pass)

Table 4.2.: GPU Forward times (in ms) of `caffe-conv` and `fft-cgemm-conv` in comparison.

Layer	out-of-place `fft-cgemm-conv` ($N = 400$)	`caffe-conv` ($N = 400$)	speed-up
conv1	99.13	100.86	$\approx 1\times$
conv2	40.33	155.78	$3.86\times$
conv3	33.33	103.43	$3.10\times$
conv4	33.11	114.84	$3.47\times$
conv5	30.98	93.20	$3.01\times$
pass	281.22	613.21	$2.18\times$

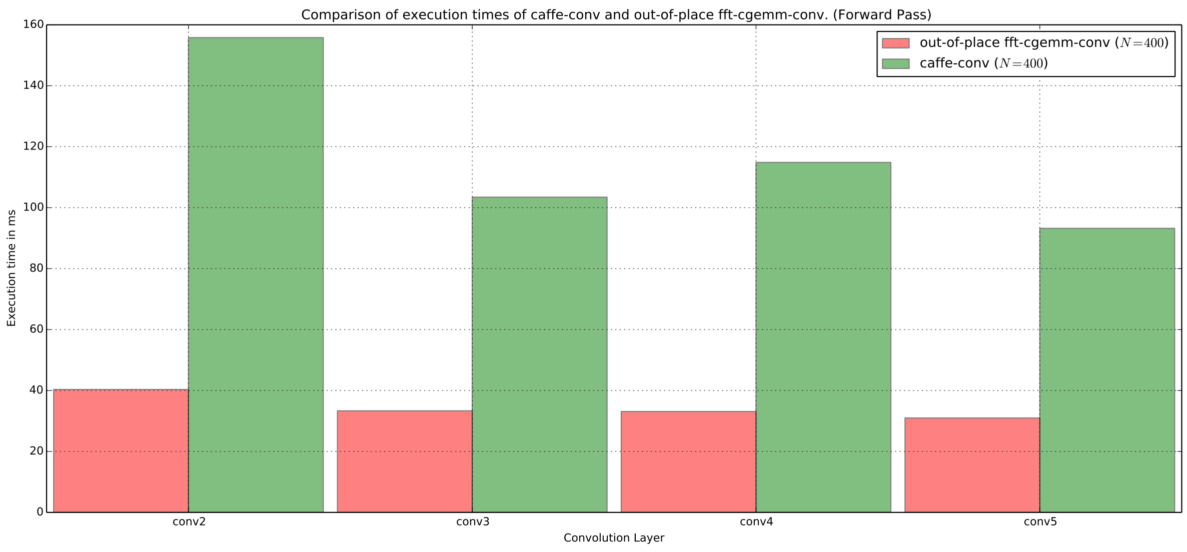


Figure 4.2.: Comparison of execution times of `caffe-conv` and out-of-place `fft-cgemm-conv` in the reference Caffe network. (GPU Forward Pass)

in-place vs. out-of-place fft-cgemm-conv We also compare the performance impact an in-place FFT has in our implementation. We see in table 4.3 that an out-of-place forward pass is about 1.05× faster in average than an in-place forward pass. The times per layer for the out-of-place `fft-cgemm-conv` and in-place `fft-cgemm-conv` are in columns two and three, respectively. Again `conv1` durations are basically identical because of the same method used in both cases. We see the differences in performance in figure 4.3.

Moreover, we check the potential memory savings possible as calculated in section 3.4.3 with the batch size of $N = 500$ used in the calculation. We first check the memory allocation needed by `caffe-conv` followed by an out-of-place `fft-cgemm-conv` and an in-place `fft-cgemm-conv`. `caffe-conv` uses constant 3737 MB of memory, the out-of-place `fft-cgemm-conv` uses 9669 MB–9880 MB of memory while the in-place `fft-cgemm-conv` uses 8076 MB–8607 MB of memory. Here we cannot confirm the theoretical predictions. There are multiple reasons for this behavior. First, we need to allocate temporary memory so we can transpose the blobs in `fft-cgemm-conv` (CuBLAS does not support an in-place transposition of matrices). Secondly, CuFFT allocates itself some more memory when performing in-place FFTs. Third, we allocate some other variables in our convolution layer. The greatest memory saving observed between the out-of-place and in-place method, therefore, is 9880 MB − 8076 MB = 1804 MB, which is almost the amount estimated in section 3.4.3.

Adjusting the batch size in fft-cgemm-conv The method of performing the fast Hadamard product clearly benefits from larger matrix dimensions. The sizes of the two matrices involved in the matrix multiplications are solely influenceable by the batch size

Table 4.3.: GPU Forward times (in ms) of out-of-place (oop) and in-place (ip) in comparison.

Layer	oop `fft-cgemm-conv` ($N = 400$)	ip `fft-cgemm-conv` ($N = 400$)	speed-up
conv1	99.13	101.69	$\approx 1\times$
conv2	40.33	44.35	$1.10\times$
conv3	33.33	36.07	$1.08\times$
conv4	33.11	37.13	$1.12\times$
conv5	30.98	30.33	$0.98\times$
pass	281.22	294.38	$1.05\times$

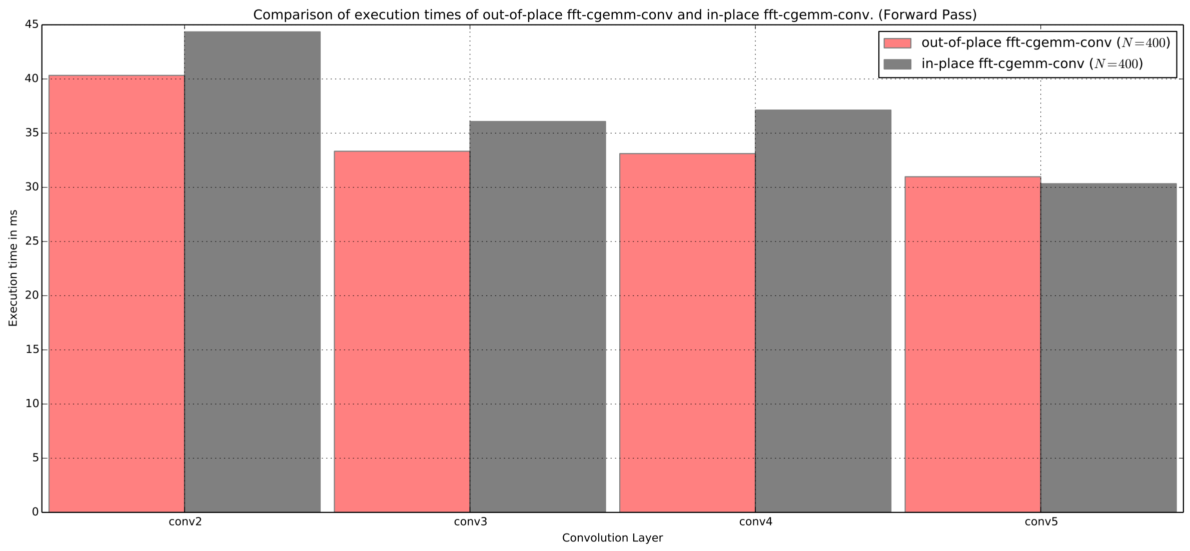


Figure 4.3.: Comparison of execution times of out-of-place `fft-cgemm-conv` and in-place `fft-cgemm-conv`. (GPU Forward Pass)

N, the number of input channels K and the number of output features N_{output}. The only parameter we can change without changing the networks architecture is the batch size N. We now compare the performance of `caffe-conv` to the out-of-place `fft-cgemm-conv` as a function of the batch size. In figure 4.4 we have visualized the results of this experiment. The graph on the top plots the average forward pass time for a single image in the batch of size $N \in [1; 400]$. We see the average pass time per image on the left y-axis and the batch size on the x-axis. On the right y-axis we see the speed-up `fft-cgemm-conv` has as a function of the batch size. We observe a maximum speed-up of 2.21× with a batch size of 384 in our experiment. On the bottom of the figure we see basically the same behavior, but the left y-axis shows the total time for the pass instead of the average time per image in the batch.

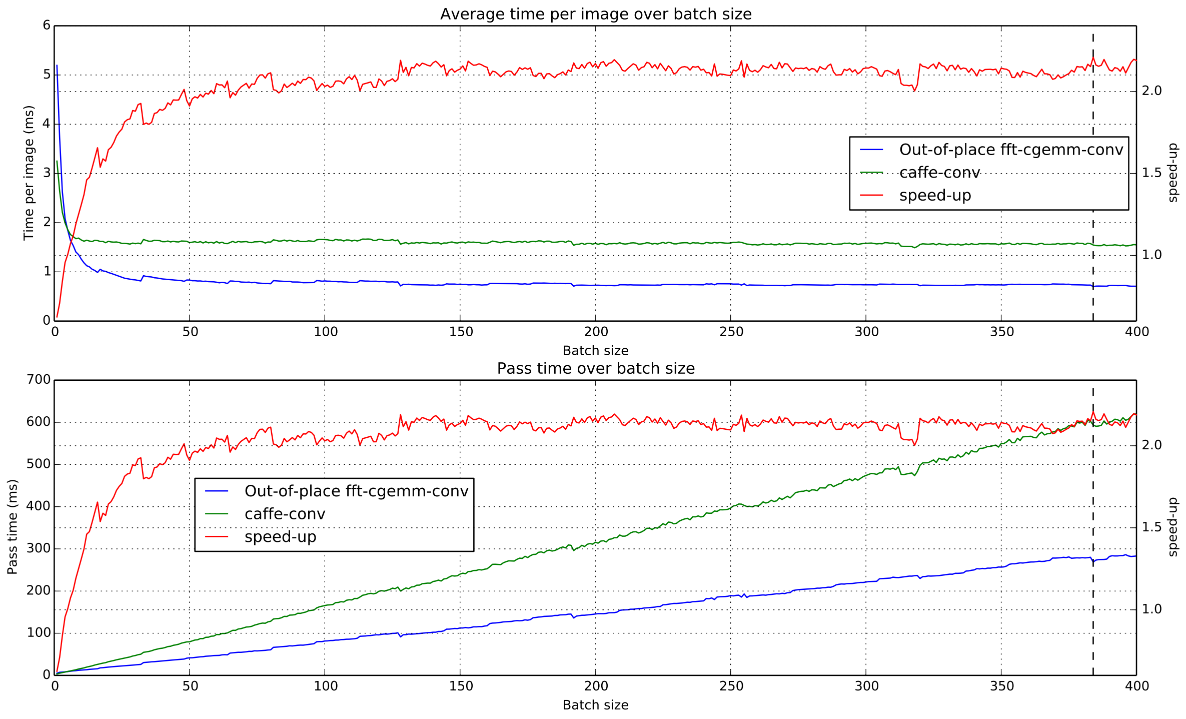


Figure 4.4.: Comparison of GPU forward times between `caffe-conv` and out-of-place `fft-cgemm-conv` with respect to the batch size.

We also compare the speed-up as a function of the batch size for every convolution layer, which uses the FFT convolution. Thereto, we show the forward time per image in a batch as a function of the batch size for convolution layers 2–5 in figure 4.5. The top left graph shows the timing result for `conv2`, the top right for `conv3`, the bottom left for `conv4` and the bottom right for `conv5`. Generally, the speed-up converges to a fixed factor in every layer beginning at a batch size of around 100. We noticed that the batch size associated with the best possible speed-up is different for every layer. The maximum speed-ups we measured are 3.96×, 3.23×, 3.63× and 3.60× for layers `conv2`, `conv3`, `conv4` and `conv5` with a batch size of 207, 254, 254 and 389, respectively. The dashed vertical line shows the batch size with the highest speed-up for each layer.

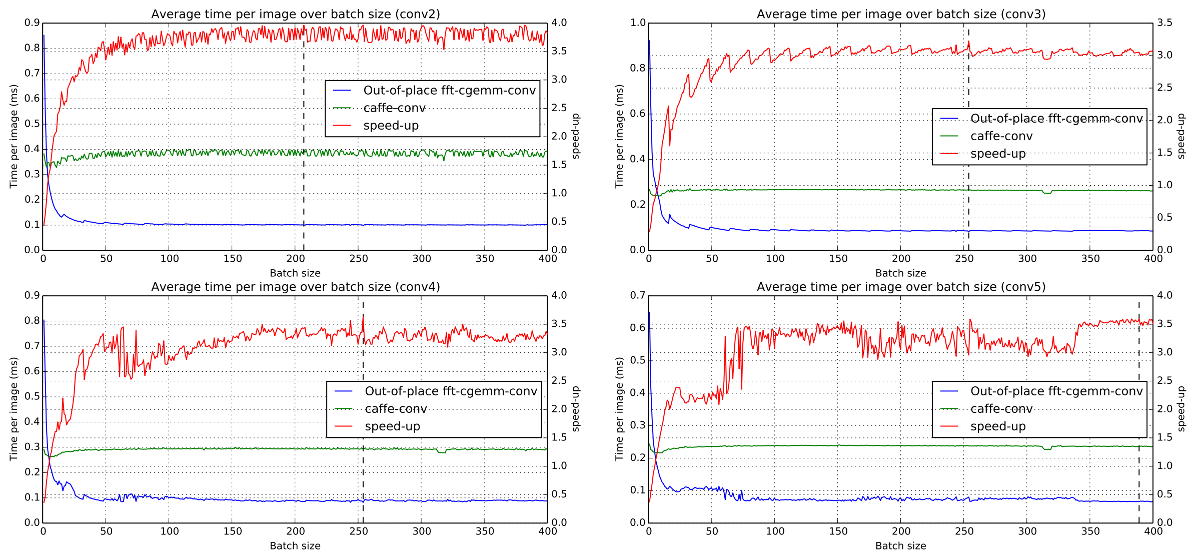


Figure 4.5.: Comparison of GPU forward times of each convolution layer between `caffe-conv` and out-of-place `fft-cgemm-conv` with respect to the batch size.

4.2.1.2. Backward Pass

Caffe's `time` mode only times the weight update step (2.15) and not a complete backward pass. Therefore, we discuss the training of the whole reference Caffe network from start to end in section 4.3. In table 4.4 we see a comparison of backward times between an out-of-place `fft-cgemm-conv`, an `fft-conv` and the traditional `caffe-conv`. Again `conv1` performs identical in all three cases. We also see that `fft-conv` does not perform better than `caffe-conv` in any layer, while `fft-cgemm-conv` performs better in convolution layers 2–5 with an overall speed-up of 1.48× for a whole backward pass with only the weight step involved. In figure 4.6 we visualize the execution time per layer for the weight update step.

4.2.2. CPU Performance

We can observe similar results in the CPU implementation. We only give comparisons of the total forward and backward pass times here, because all values behave similarly to those on the GPU reported in section 4.2.1. We also omit measuring the performance for varying batch sizes.

Table 4.4.: GPU Backward times (in ms) of out-of-place (oop) `fft-cgemm-conv`, `fft-conv` and `caffe-conv` in comparison.

Layer	oop `fft-cgemm-conv` ($N = 400$)	`fft-conv` ($N = 400$)	`caffe-conv` ($N = 400$)	speed-up
conv1	108.44	112.27	110.92	$\approx 1\times$
conv2	44.42	134.71	120.88	$2.72\times$
conv3	39.82	209.70	46.34	$1.16\times$
conv4	36.96	157.84	58.00	$1.57\times$
conv5	28.35	107.11	53.99	$1.90\times$
pass	277.50	741.56	409.96	$1.48\times$

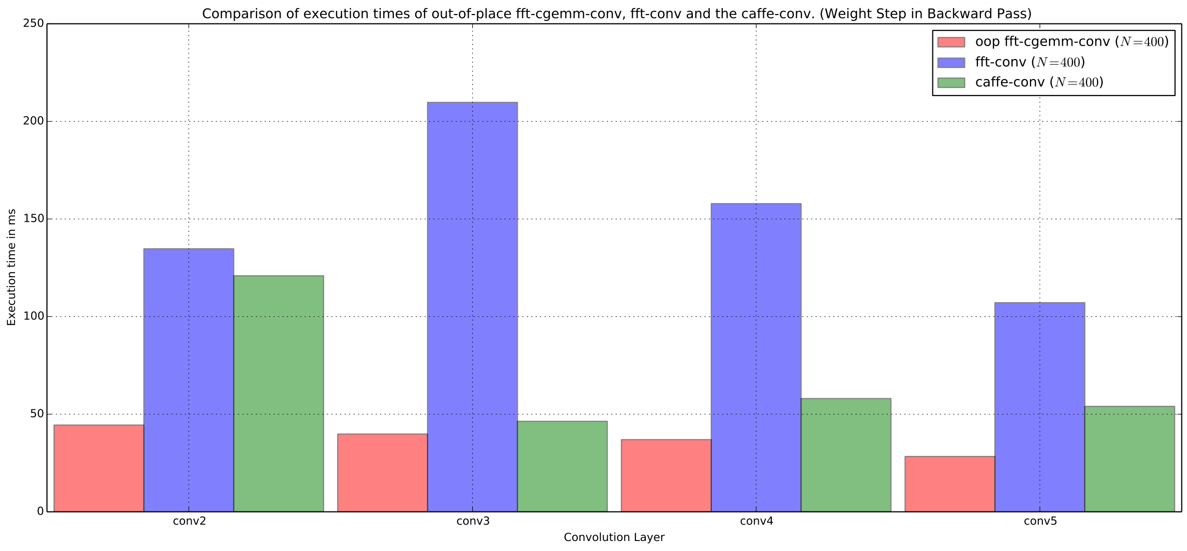


Figure 4.6.: Comparison of execution times of out-of-place `fft-cgemm-conv`, `fft-conv` and `caffe-conv`. (Weight step in backward pass)

4.2.2.1. Forward Pass

We used the same batch size of $N = 400$ for the timings on the CPU. We also ran each test 200 times like we did for the GPU.

caffe-conv vs. fft-conv We evaluate the simple Hadamard product with `fft-conv`. The three loops we parallelized in section 3.4.1 cannot take as much advantage on a CPU as on a GPU, which uses thousands of simultaneous running threads. Altogether, we have a slow-down of 3.1× when using `fft-conv`. A single forward pass with the given batch sizes takes 21,663.90 ms and 7004.09 ms for `fft-conv` and `caffe-conv` respectively. Also note that we perform the same tasks 11.42× and 31.23× faster on our GPU.

caffe-conv vs. out-of-place fft-cgemm-conv Although the CPU implementation is several times slower than the GPU version, the measured speed-up of the out-of-place `fft-cgemm-conv` is similar to the speed-up we observed on the GPU. We timed forward pass times of 4000.35 ms and 7004.09 ms for the out-of-place `fft-cgemm-conv` and `caffe-conv` respectively. This is a speed-up of 1.75× in contrast to 2.18× on the GPU.

in-place vs. out-of-place fft-cgemm-conv We found out that it almost does not matter if we perform an out-of-place or an in-place version of `fft-cgemm-conv` on the CPU. A forward pass for the in-place took 3985.81 ms in contrast to 4000.35 ms, which is even a little faster but roughly no speed-up at all considering measuring inaccuracies.

4.2.2.2. Backward Pass

The backward pass on the CPU (again we were only able to measure the weight step) is slightly faster, when we used `fft-cgemm-conv`. The speed-up we could find out was a little better than on the GPU. For the out-of-place `fft-cgemm-conv` we measured a backward time of 3115.69 ms, while `caffe-conv` took 5516.80 ms to finish. Thus, the speed-up on the CPU is 1.77× versus 1.48× on the GPU. We also checked execution times for `fft-conv` which performs 8.84× slower than the traditional Caffe convolution.

4.3. Training the Reference Caffe Network

We could only measure timings for the weight step (2.15) in section 4.2.1.2, because Caffe's `time` mode does not measure time for the backpropagation step (2.16). Therefore, we trained the reference Caffe network both with our convolution layer using the in-place `fft-cgemm-conv` and the traditional `caffe-conv`. The reference Caffe network needs about 310,000 iterations to perform best. In each iteration the training step consists of a forward pass followed by a backward pass, where 256 images of the ILSVRC 2012 training set are forwarded in a batch. We validate the learned weights in a network every 2500 iterations over the ILSVRC 2012 validation set. While training we decrease the learning rate by a factor of 10 every $100,000$ iterations. A trained reference Caffe network can reach a validation accuracy of around 57.412% according to the pre-trained model shipped with Caffe (The model is snapshotted at iteration 310,000 with a validation accuracy of 56.874% and the best validation accuracy of 57.412% was at iteration 313,000.). We reached a validation accuracy of 56.342% when we trained with the in-place `fft-cgemm-conv` method and 57.01% with `caffe-conv` after exactly 310,000 iterations. We trained the reference Caffe network on the GPU.

We compare the times both trainings needed to reach iteration 310,000. `caffe-conv` needed 84.84 h to reach iteration 310,000 while the in-place `fft-cgemm-conv` needed only 47.52 h to finish the training. Wee see that our method (note we used the slower in-place version) has a speed-up of about 1.79× when we train the reference Caffe network. This is slower than in a forward pass, but we need to Fourier transform the weights in every single iteration, because the weights are adjusted while training. We visualized the training process with both methods in figure 4.7, where we see validation accuracy over time on the left side. Here we see how much faster our method reaches a high validation accuracy in contrast to `caffe-conv`. We also notice that both methods learn things the same way, as the accuracy increases considerably on iterations 100,000, 200,000 and 300,000 when the learning rate is decreased. On the right side of the figure we plotted the training duration respective to the training iteration. The dashed line visualizes training iteration 310,000 of the in-place `fft-cgemm-conv` training.

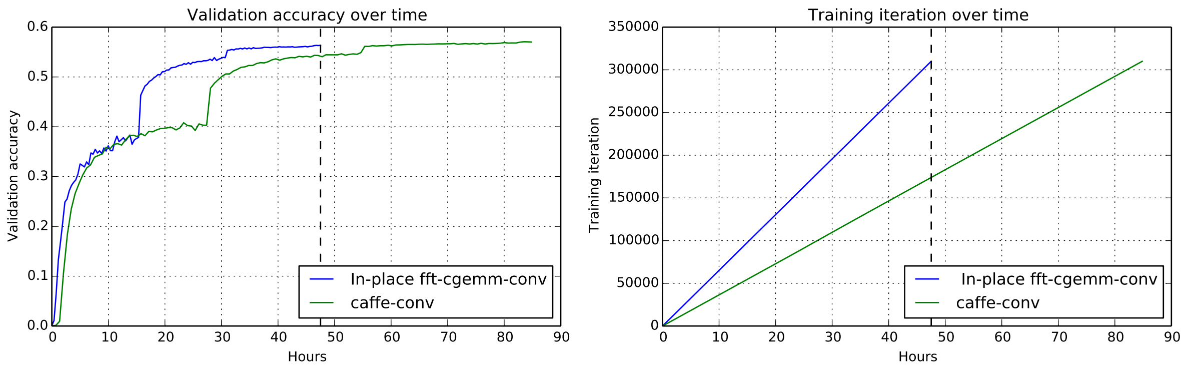


Figure 4.7.: Comparison of `caffe-conv` and in-place `fft-cgemm-conv` training times of the reference Caffe network.

In figure 4.8 we see the training loss over the iteration number and over the training time respectively. On top we again can notice that both methods do the same thing and both method shortly asymptotically converge to the same loss. On the bottom we plotted the loss respective to the training time, where we can see one more time that our method does the same thing considerably faster. We visualize the end of the in-place `fft-cgemm-conv` training with a dashed line again.

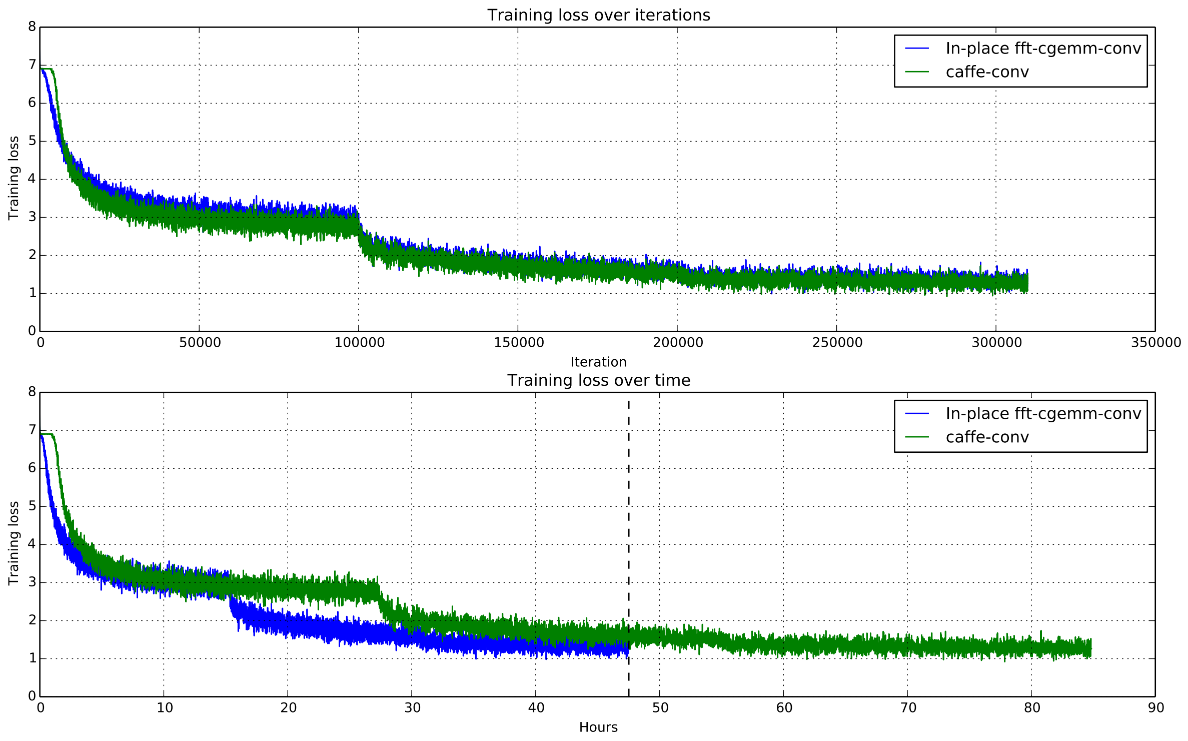


Figure 4.8.: Comparison of `caffe-conv` and in-place `fft-cgemm-conv` GPU training loss in respect to the iteration number and training time while training the reference Caffe network.

4.4. Designing FFT-Exploiting Layers

We now design a layer exploiting properties of the FFT. The FFT sizes we calculate in our layer (3.2) sometimes leave room to increase the kernel size without increasing the FFT size. Therefore, greater kernel sizes do not need any additional computation time in the frequency domain. We used the GPU implementation to compare results of the custom designed layers.

First, we altered the first convolution layer of the reference Caffe network to demonstrate huge speed-ups by exploiting FFT properties. We changed the stride parameter from $s = 4$ to $s = 1$ and altered the kernel size in our experiments to range from 11 to 29 with a step size of 2. So, we used the kernel sizes $K_h = K_w \in [11; 13; 15; 17; 19; 21; 23; 25; 29]$. For the kernel sizes 11 and 13 the FFT size is 240, while it is 256 for the other kernel sizes. We know that the computational complexity of the convolution in the time domain increases quadratically with the kernel size, while it does not in the frequency domain as long as the FFT size stays constant. We confirm this in table 4.5 where we tested our custom layer with a batch size of $N = 50$ (We could not fit a greater batch size into GPU memory).

Table 4.5.: GPU Forward times (in ms) of out-of-place (oop) `fft-cgemm-conv` and `fft-conv` with varying kernel size within the first custom convolution layer in comparison.

kernel size	oop `fft-cgemm-conv` ($N = 50$)	`caffe-conv` ($N = 50$)	speed-up
11	71.69	128.69	1.80×
13	71.84	182.77	2.54×
15	76.58	235.64	3.08×
17	78.51	294.44	3.75×
19	77.35	357.61	4.62×
21	77.87	427.94	5.50×
23	74.48	490.61	6.59×
25	77.22	583.59	7.56×
27	76.56	667.28	8.72×
29	76.40	755.99	9.90×

This table shows almost the same execution time for kernel sizes of $11, 13$ and $15, 17, 19, 21, 23, 25, 27, 29$ respectively for the out-of-place `fft-cgemm-conv`. In contrast `caffe-conv` takes longer for greater kernel sizes. For a kernel size of 29 we achieve a speed-up of almost $10\times$. Note that we also have a speed-up for a kernel size of 11 in contrast to the our theoretical predictions here. We did not use the FFT convolution in the first convolution layer of the reference Caffe network before, because it did not pay off. We solely have a speed-up here, because we set $s = 1$. The Caffe convolution in the time-domain computes the convolution at every $s = 4$-th position in both dimensions. In the frequency domain this is not possible since we loose all spatial information after computing the frequency domain representation. Therefore, the first convolution layer of the reference Caffe network does not promise any speed-ups in contrast to our custom designed layer. We visualize the speed-ups as a function of the kernel size in figure 4.9. The forward time is on the left y-axis, which has a logarithmic scale. We plot the speed-up on the right y-axis. On the x-axis we see the kernel size.

We constructed a second custom convolution layer which has an input of size $55 \times 55 \times 96$ on which we tested the kernel sizes $K_h = K_w \in \{2n + 1 : 0 \leq n \leq 19\}$. We set both the stride and padding to 0 in this layer, while we now use a batch size of $N = 200$. The FFT sizes are $64, 80$ and 96 for the kernel sizes $\{2n + 1 : 0 \leq n \leq 4\}$, $\{2n + 1 : 5 \leq n \leq 12\}$ and $\{2n + 1 : 13 \leq n \leq 19\}$ respectively. Note the number of positions we can convolve varies, because we do not adapt the padding parameter accordingly. Thus, we do not get a linear speed up with increasing kernel size. We visualize the results of our experiment

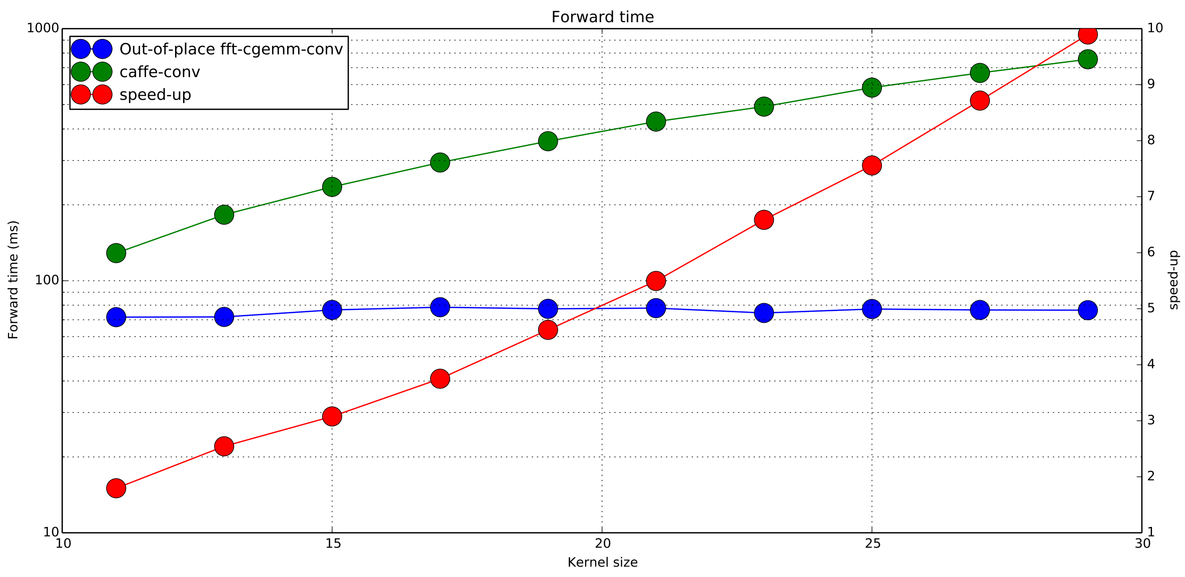


Figure 4.9.: Comparison of `caffe-conv` and in-place `fft-cgemm-conv` forward times when adjusting the kernel size in the first custom layer.

in table 4.6 and figure 4.10 with the same format as the previous table and previous figure. Here we see that we can achieve huge speed-ups of approximately 20× with a kernel size of 31 in this layer, while the FFT convolution is not as beneficial for small kernel sizes.

We can use this property in designing new network architectures that take advantage of bigger kernels. We have yet to find out if greater kernel sizes increase the validation accuracy or improve other factors of a DCNN.

4.5. Testing for Correctness

The most important fact is if our `ConvolutionLayerFFT` actually calculates a correct result and returns the same result as a `ConvolutionLayer` every time. We ensure the correctness by implementing unit tests of our own and also compare the validation accuracy of the FFT convolution with the Caffe convolution. We do both assurances for every different method (`fft-conv`, out-of-place `fft-cgemm-conv` and in-place `fft-cgemm-conv`) we implemented on the GPU as well as on the CPU.

Table 4.6.: GPU Forward times (in ms) of out-of-place (oop) `fft-cgemm-conv` and `fft-conv` with varying kernel size within the second custom convolution layer in comparison.

kernel size	oop `fft-cgemm-conv` ($N = 200$)	`caffe-conv` ($N = 200$)	speed-up
1	85.51	15.35	0.18×
3	85.36	78.65	0.92×
5	88.52	177.05	2.00×
7	88.43	375.30	4.24×
9	88.00	448.47	5.10×
11	132.12	821.54	6.22×
13	132.00	897.35	6.80×
15	131.76	1135.73	8.62×
17	130.47	1845.20	14.14×
19	130.00	1592.67	12.25×
21	129.75	1864.20	14.37×
23	129.30	2187.03	16.91×
25	128.45	2471.96	19.25×
27	184.31	2566.16	13.92×
29	183.57	2930.70	15.96×
31	183.07	3644.34	19.91×
33	185.58	3054.51	16.46×
35	184.75	3406.56	18.44×
37	184.70	3313.29	17.94×
39	185.01	3025.50	16.35×

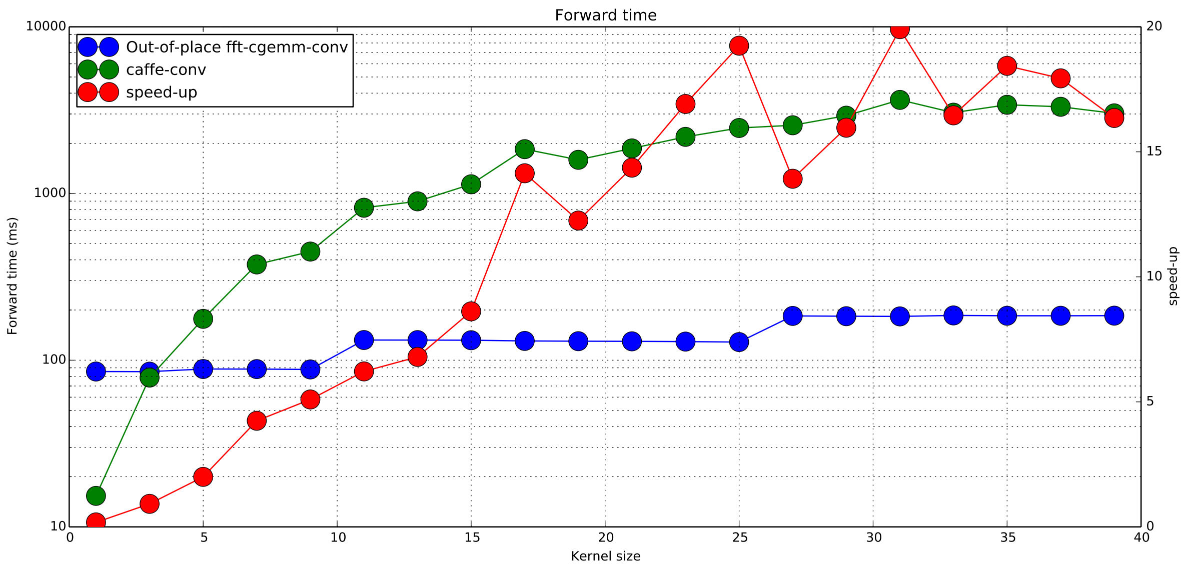


Figure 4.10.: Comparison of `caffe-conv` and in-place `fft-cgemm-conv` GPU forward times when adjusting the kernel size in the second custom layer.

4.5.1. Unit tests

The developers of Caffe assured the correctness of their layers by implementing unit tests for every layer. In the special case of the convolution layer they implemented a reference convolution method that performs convolutions with random values. The unit tests examine many different combinations between convolution parameters (kernel size, stride, padding, group...) for the forward pass. For the backpropagation, unit tests ensure the implementation correctness by numerically calculating a gradient [14] and comparing it with the backward step's result. The mathematical definition for a derivation can be expressed like this:

$$\frac{\partial f(x,y)}{\partial x} = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

Note that h is very small here. In the unit tests we modify weights by a small amount h, perform a forward pass with those weights and compare the difference with the backward pass' result.

We duplicated those unit tests and expanded them to test many different ways our own `ConvolutionLayerFFT` can compute the convolution. This unit tests compare the layer's output and compare it with the reference values for CPU and GPU and both single-precision and double-precision values. All tests passed on our layer.

4.5.2. Forward Pass on Validation Set

We also used the ILSVRC 2012 validation set and compared the validation accuracy with the validation accuracy reported by `caffe-conv`. We tested on all 50,000 images and found both methods to yield the exact same result of 56.874% on the reference Caffe network.

Afterward, we used our newly FFT-trained reference Caffe network and tested the validation set on the trained network both with `caffe-conv` layers and layers using `fft-cgemm-conv`. Both methods reported the same validation accuracy of 56.342% we reported earlier.

5. Conclusion and Future Work

5.1. Summary

In this master thesis, we used the well-known method of pointwise multiplying in the frequency domain to implement a faster method of performing a convolution within the Caffe framework. Because most recent and also larger DCNNs need more time to calculate results even on contemporary hardware, we developed a convolution layer that tries to mitigate these limitations.

We began this thesis by introducing the convolution operation and explaining how the convolution is performed in a Caffe convolution layer in chapter 2. We then described a typical DCNN which we used as an example to explain the theoretical groundwork. Afterward, we gave an introduction into Fourier transforms and described how the convolution in the frequency domain is done by using the convolution theorem through pointwise multiplication. Thereupon we suggested an even faster way of evaluating the pointwise product in the frequency domain. Furthermore, we estimated whether a convolution in the frequency domain pays off on a typical DCNN by calculating the number of arithmetic operations for both the Caffe convolution and the frequency domain convolution. Because these results were promising, we examined whether the training operation of a DCNN can also be expressed through convolution operations to speed up the – very time-consuming – backpropagation.

In chapter 3 we described important details we used in the implementation of our own FFT convolution layer. We started by describing how the Caffe framework is constructed and handles its input data. We then described how the convolution is performed in Caffe and which pitfalls we have to avoid to develop our own convolution layer. Afterward, we started implementing our own frequency domain convolution layer. We enumerated the libraries for the FFT and matrix multiplications we used both for the GPU and CPU implementation. After that, we made sure to preserve the compatibility with the Caffe convolution layer and other layers by transforming results back in the time domain. Thereupon, we prepared the data in the layer for taking the FFT most efficiently followed by computing the pointwise product in the frequency domain. After that, we gave details on how to transform the frequency representation back to the time domain and normalize the result. We then described how the full forward and backward pass are implemented.

Finally, we gave details on how to optimize the process even more and outlined some special implementation details of a GPU implementation.

At the end of our thesis, we evaluated our frequency convolution layer in chapter 4 by using various test scenarios on the GPU as well as on the CPU. We confirmed the theoretical assumptions made before on the typical DCNN. On this network, we achieved speed-ups of more than $2\times$ without altering the network's architecture in our favor. We found out that the speed-up is higher with an increasing batch size. After that, we tested training the typical network as a whole, both with our method and Caffe's method. We found out that our method finished training in almost $1.8\times$ less time than Caffe's convolution layer. We then constructed two layers exploiting properties of the FFT in our favor which yielded speed-ups of $10\times$ and $20\times$ respectively. Finally, we made sure our convolution layer also works correctly and returns the same results as Caffe's convolution layer by duplicating Caffe's unit tests for our layer.

Alongside with this thesis, we supply an implementation. Our implementation extends the capabilities of the Caffe framework in the form of an additional `ConvolutionLayerFFT` that uses the FFT to compute the convolution. We also provide the implementation on Github[1].

5.2. Future work

We will focus on reducing the memory footprint in future work. We found that allocating additional memory required for Fourier transforms needs to be done only once. We allocate additional memory only for the convolution layer that consumes most memory. According to the reference Caffe network and table 3.4 we only allocate memory for `conv2` and reuse this allocated memory area in `conv3`, `conv4` and `conv5`. By doing so, we can save further 1368 MB of memory, which is equal to 64 % of the memory we allocated for an in-place FFT. Additionally, we can save even more memory by just retaking the Fourier transform of the weights every forward pass, however, sacrificing computation performance in return. Using this method, we hope to fit larger DCNNs in the restricted memory of contemporary GPUs.

We also want to explore other network architectures taking advantage of FFT properties such as weights with bigger kernel sizes. We hope to find some interesting properties DCNNs with bigger kernel sizes have.

[1]Our implementation is available on `https://github.com/philm5/caffe`.

6. List of Figures

7. List of Tables

Bibliography

[1] C.M. Bishop. *Pattern Recognition and Machine Learning.* Information Science and Statistics. Springer, 2006. ISBN: 9780387310732.

[2] Ronald N Bracewell. *The Fourier transform and its applications.* McGraw-Hill electrical and electronic engineering series. McGraw-Hill, 1986. ISBN: 9780070070158.

[3] Kumar Chellapilla, Sidd Puri, and Patrice Simard. "High performance convolutional neural networks for document processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition.* Suvisoft. 2006.

[4] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[5] Intel Corporation. *Intel® Math Kernel Library.* URL: `https://software.intel.com/en-us/intel-mkl`.

[6] NVIDIA Corporation. *NVIDIA® CuBLAS library.* URL: `http://docs.nvidia.com/cuda/cublas/`.

[7] NVIDIA Corporation. *NVIDIA® CuFFT library.* URL: `http://docs.nvidia.com/cuda/cufft/`.

[8] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[9] Andrew Gibiansky. *Convolutional Neural Networks.* 2014. URL: `http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/` (visited on 12/17/2015).

[10] Nicholas J Higham. "Exploiting fast matrix multiplication within the level 3 BLAS". In: *ACM Transactions on Mathematical Software (TOMS)* 16.4 (1990), pp. 352–368.

[11] Google Inc. *Google Protocol Buffers.* URL: `https://developers.google.com/protocol-buffers/`.

[12] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[13] Steven G. Johnson and Matteo Frigo. "Implementing FFTs in Practice". In: *Fast Fourier Transforms.* Ed. by C. Sidney Burrus. Rice University, Houston TX: Connexions, 2008. Chap. 11. URL: `http://cnx.org/content/m16336/`.

[14] Andrej Karpathy. *Hacker's guide to Neural Networks.* 2015. URL: `http://karpathy.github.io/neuralnets/`.

[15] David Kirk et al. "NVIDIA CUDA software and GPU parallel computing architecture". In: *ISMM.* Vol. 7. 2007, pp. 103–104.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[17] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[18] Richard G Lyons. *Understanding Digital Signal Processing.* Pearson Education International, 2011. ISBN: 9780132119375.

[19] Michael Mathieu, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through FFTs". In: *arXiv preprint arXiv:1312.5851* (2013).

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5.* 2015. URL: `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[21] Prof. Brad Osgood. *Lecture Notes for Stanford EE261: The Fourier Transform and its Applications.* 2007. URL: `https://see.stanford.edu/materials/lsoftaee261/book-fall-07.pdf`.

[22] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[23] S.W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Pub., 1997. ISBN: 9780966017632.

[24] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356.

[25] Nicolas Vasilache et al. "Fast convolutional nets with fbfft: A GPU performance evaluation". In: *arXiv preprint arXiv:1412.7580* (2014).

[26] Zhang Xianyi, Wang Qian, and Zaheer Chothia. "OpenBLAS". In: (2012). URL: `http://www.openblas.net`.

A. Derivation of the Convolution Theorem

This appendix gives the derivation of the convolution theorem. The derivation we give is based on [21]. The convolution theorem states that a multiplication in frequency domain is equal to the convolution in the time domain. So we take the product of Fourier transforms of $f(x)$ and $g(t)$:

$$\hat{f}(m)\hat{g}(m) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi xm}\ dx \int_{-\infty}^{\infty} g(t)e^{-i2\pi tm}\ dt$$

Note that we use two different variables for integration so we can combine the product into a iterated integral:

$$\begin{aligned}\int_{-\infty}^{\infty} f(x)e^{-i2\pi xm}\ dx \int_{-\infty}^{\infty} g(t)e^{-i2\pi tm}\ dt &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-i2\pi xm}e^{-i2\pi tm} f(x)g(t)\ dx\ dt \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-i2\pi(x+t)m} f(x)g(t)\ dt\ dx \\ &= \int_{-\infty}^{\infty}\Big[\int_{-\infty}^{\infty} e^{-i2\pi(x+t)m} g(t)\ dt\Big] f(x)\ dx\end{aligned}$$

We substitute $x+t$ by u and therefore get $t = u - x$ and $dt = du$. Afterward, we switch the order of integration:

$$\begin{aligned}\int_{-\infty}^{\infty}\Big[\int_{-\infty}^{\infty} e^{-i2\pi(x+t)m} g(t)\ dt\Big] f(x)\ dx &= \int_{-\infty}^{\infty}\Big[\int_{-\infty}^{\infty} e^{-i2\pi um} g(u-x)\ du\Big] f(x)\ dx \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-i2\pi um} g(u-x)f(x)\ du\ dx \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-i2\pi um} g(u-x)f(x)\ dx\ du \\ &= \int_{-\infty}^{\infty} e^{-i2\pi um}\underbrace{\Big[\int_{-\infty}^{\infty} g(u-x)f(x)\ dx\Big]}_{h(u)}\ du\end{aligned}$$

We see that $h(u) = \int_{-\infty}^{\infty} g(u-x)f(x)\ dx = \int_{-\infty}^{\infty} f(x)g(u-x)\ dx$ is nothing other than the continuous convolution as defined in [2]. We can also see that the Fourier transform of $h(u)$ is taken:

$$\begin{aligned}\int_{-\infty}^{\infty} e^{-i2\pi um}\Big[\int_{-\infty}^{\infty} g(u-x)f(x)\ dx\Big]\ du &= \int_{-\infty}^{\infty} e^{-i2\pi um}h(u)\ du \\ &= \int_{-\infty}^{\infty} h(u)e^{-i2\pi um}\ du \\ &= \hat{h}(m)\end{aligned}$$

Therefore, we can conclude that the multiplication in the frequency domain is the same as taking the Fourier transform of the convolution in time domain:

$$\Rightarrow \hat{h}(m) = \mathcal{F}(f(x) * g(x))[m] = \mathcal{F}(f(x))[m] \cdot \mathcal{F}(g(x))[m] = \hat{f}(m) \cdot \hat{g}(m).$$

Acknowledgments

At first, I want to thank Prof. Dr. Lienhart for enabling me to write my master thesis at the Multimedia Computing and Computer Vision Lab and giving me suggestions throughout the thesis.

I also want to thank my supervisor Christian Eggert, whom I could always bother with questions, for helpful remarks and discussions. Furthermore, I am very grateful for the other staff members, who always had a friendly ear and advice.

Finally, I want to express my gratitude for my parents, who always supported me financially and, therefore, enabling me to focus on my studies.