EFFICIENT DEEP LEARNING OF 3D STRUCTURAL BRAIN MRIS FOR MANIFOLD LEARNING AND LESION SEGMENTATION WITH APPLICATION TO MULTIPLE SCLEROSIS

by

TOM BROSCH

Dipl.-Ing., Otto-von-Guericke-Universität Magdeburg, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Biomedical Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

January 2016

Abstract

Deep learning has shown great promise in recent years for various non-medical and medical problems. However, the size of images has been a burden and problems had to be mapped to either small 2D or 3D patches, or 2D images. This thesis analysis the potential of deep learning to solve medical image analysis problems using entire 3D volumes. A major part of the thesis was the development of a training method for deep learning models that facilitates the training on entire 3D volumes. We have applied the new training method to solve different medical image analysis problems, such as manifold learning for biomarker discovery for AD using raw images, and MS using deformation fields, and lesion masks. A second part of the thesis was the development of an MS lesion segmentation method using multimodal MR images. We evaluated the runtime improvements against other state-of-the-art methods. Furthermore, we showed the clinical value of our approach on medical problems.

Table of Contents

List of Tables

List of Figures

1 Training of Convolutional Models in the Frequency Domain

1.1 Related Work

Deep learning has traditionally been computationally expensive and advances in training methods have been the prerequisite for expanding its application to a variety of image classification problems. The development of layer-wise training methods (**?**) greatly improved the efficiency of the training of deep belief networks (DBNs), which has made feasible the use of large sets of small images (e.g. 28×28), such as those used for hand-written digit recognition. Subsequently, new directions for speeding up the training of deep models were opened with the advance of programmable graphics cards (GPUs), which can perform thousands of operations in parallel. **?** demonstrated that by using graphics cards, training of restricted Boltzmann machines (RBMs) on small image patches (e.g. 24×24) can be performed up to 70 times faster than on the CPU, facilitating the application to larger training sets. However, the number of trainable weights of a DBN increases greatly with the resolution of the training images, which can make training on large images impracticable. In order to scale DBNs to high-resolution images, **??** introduced the convolutional deep belief network (convDBN), a deep generative model

that uses weight sharing to reduce the number of trainable weights. They showed that a convDBN can be used to classify images with a resolution up to 200×200 pixels. To speed up the training of convolutional neural networks (CNNs) on high-resolution images, **?** replaced traditional convolutions of the first layer of their CNN with strided convolutions, a type of convolution that shifts the filter kernel as a sliding window with a fixed step size or stride greater than one. Through the use of strided convolutions, the number of hidden units in each convolutional layer is greatly reduced, which reduces both training time and required memory. Using a highly optimized GPU implementation of convolutions, they were able to train a CNN on images with a resolution of 256×256 pixels, achieving state-of-the-art performance on the ILSVRC-2010 and ILSVRC-2012 competitions (**?**). An alternative approach for calculating convolutions was proposed by **?** who sped up the training of CNNs by calculating convolutions between batches of images and filters using fast Fourier transforms (FFTs), albeit at the cost of additional memory required for storing the filters. Recently, NVIDIA have released a library called cuDNN **?** that provides GPU-optimized implementations of 2D and 3D convolutions among other operations that are frequently used to implement deep learning methods, which further reduces training times compared to the previous state-of-the-art.

1.2 Overview

In this chapter, we detail our training algorithm and GPU implementation in full, with a thorough analysis of the running time on high-resolution 2D images (512×512) and 3D volumes ($128 \times 128 \times 128$), showing speed-ups of up to 8-fold and 200-fold, respectively for training RBMs and a 7-fold speed up for computing key operations for the training CNNs compared to cuDNN. The proposed method performs training in the frequency

domain, which replaces the calculation of time-consuming convolutions with simple element-wise multiplications, while adding only a small number of FFTs. In contrast to similar FFT-based approaches (e.g., **?**), our method does not use batch processing of the images as a means to reduce the number of FFT calculations, but rather minimizes FFTs even when processing a single image, which significantly reduces the required amount of scarce GPU memory. In addition, we formalize the expression of the strided convolutional DBN (sconvDBN), a type of convDBN that uses strided convolutions to speed up training and reduce memory requirements, in terms of stride-1 convolutions, which enables the efficient training of sconvDBNs in the frequency domain.

1.3 Algorithm

1.3.1 Training in the Spatial Domain

Before we look at the training algorithm of convRBMs in the frequency domain, let us revise the basic steps for training in the spatial domain. The weights and bias terms of a convRBM can be learned by CD. During each iteration of the algorithm, the gradient of each parameter is estimated and a gradient step with a fixed learning rate is applied. The gradient of the filter weights can be approximated by

$$\Delta \mathbf{w}^{(ij)} \approx \frac{1}{N}(\mathbf{v}_n^{(i)} * \tilde{\mathbf{h}}_n^{(j)} - \mathbf{v}_n'^{(i)} * \tilde{\mathbf{h}}_n'^{(j)}) \tag{1.1}$$

where $\mathbf{v}_n, n \in [1, N]$ are images from the training set, $\mathbf{h}_n^{(j)}$ and $\mathbf{h}_n'^{(j)}$ are samples drawn from $p(\mathbf{h}^{(j)} \mid \mathbf{v}_n)$ and $p(\mathbf{h}^{(j)} \mid \mathbf{v}_n')$, and $\mathbf{v}_n'^{(i)} = \mathbb{E}[\mathbf{v}^{(i)} \mid \mathbf{h}_n]$. To apply the model to real-valued data like certain types of images, the visible units can be modeled as Gaussian units. When

the visible units are mean–centered and standardized to unit variance, the expectation of the visible units is given by

$$\mathbb{E}[\mathbf{v}^{(i)} \mid \mathbf{h}] = \sum_{j=1}^{N_\mathrm{k}} \mathbf{w}^{(ij)} * \mathbf{h}^{(j)} + b_i \tag{1.2}$$

A binary hidden unit can only encode two states. In order to increase the expressive power of the hidden units, we use noisy rectified linear units as the hidden units, which have been shown to improve the learning performance of RBMs (**?**). The hidden units can be sampled with

$$\mathbf{h}^{(j)} \sim \max(0, \mu^{(j)} + \mathcal{N}(0, \mathrm{sigm}(\mu^{(j)}))) \tag{1.3}$$

$$\mu^{(j)} = \sum_{i=1}^{N_\mathrm{c}} \tilde{\mathbf{w}}^{(ij)} * \mathbf{v}^{(i)} + c_j \tag{1.4}$$

where $\mathcal{N}(0, \sigma^2)$ denotes Gaussian noise. The learning algorithm in the spatial domain is summarized in Figure 1.1a.

1.3.2 Training in the Frequency Domain

The computational bottleneck of the training algorithm in the spatial domain is the calculation of convolutions, which needs to be performed $5 \times N_\mathrm{c} \times N_\mathrm{k}$ times per iteration. To speed up the calculation of convolutions, we perform training in the frequency domain, which maps the convolutions to simple element-wise multiplications. This is especially important for the training on 3D images due to the relatively large number of weights of a 3D kernel compared to 2D. To minimize the number of Fourier transforms, we map all operations needed for training to the frequency domain whenever possible, which allows the training algorithm to stay almost entirely in the frequency domain. All of the scalar

input : Images $\mathcal{D} = \{\mathbf{V}_n \mid n \in [1, N]\}$
output: Weights gradient $\Delta\mathbf{w}$

$\Delta\mathbf{w} = 0$
foreach image $\mathbf{v} \in \mathcal{D}$ **do**
  $\mathbf{v}' = 0$
  for $j = 1$ **to** N_k **do**
    $\mu^{(j)} = \sum_{i=1}^{N_\mathrm{c}} \tilde{\mathbf{w}}^{(ij)} * \mathbf{v}^{(i)} + c_j$
    $\mathbf{h}^{(j)} \sim \max(0, \mu^{(j)} + \mathcal{N}(0, \mathrm{sigm}(\mu^{(j)})))$
    for $i = 1$ **to** N_c **do**
      $\Delta\mathbf{w}^{(ij)} = \Delta\mathbf{w}^{(ij)} + \tilde{\mathbf{h}}^{(j)} * \mathbf{v}^{(i)}$
      $\mathbf{v}'^{(i)} = \mathbf{v}'^{(i)} + \mathbf{w}^{(ij)} * \mathbf{h}^{(j)}$
  $\forall i\colon \mathbf{v}'^{(i)} = \mathbf{v}'^{(i)} + b_i$
  for $j = 1$ **to** N_k **do**
    $\mu'^{(j)} = \sum_{i=1}^{N_\mathrm{c}} \tilde{\mathbf{w}}^{(ij)} * \mathbf{v}'^{(i)} + c_j$
    $\mathbf{h}'^{(j)} \sim \max(0, \mu'^{(j)} + \mathcal{N}(0, \mathrm{sigm}(\mu^{(j)})))$
    for $i = 1$ **to** N_c **do**
      $\Delta\mathbf{w}^{(ij)} = \Delta\mathbf{w}^{(ij)} - \tilde{\mathbf{h}}'^{(j)} * \mathbf{v}'^{(i)}$

(a) Training in the spatial domain

input : Images $\hat{\mathcal{D}} = \{\hat{\mathbf{v}}_n \mid n \in [1, N]\}$
output: Weights gradient $\Delta\hat{\mathbf{w}}$

$\Delta\hat{\mathbf{w}} = 0$
foreach image $\hat{\mathbf{v}} \in \hat{\mathcal{D}}$ **do**
  $\hat{\mathbf{v}}' = 0$
  for $j = 1$ **to** N_k **do**
    $\hat{\mu}^{(j)} = \sum_{i=1}^{N_\mathrm{c}} \overline{\hat{\mathbf{w}}^{(ij)}} \cdot \hat{\mathbf{v}}^{(i)}$
    $\mu^{(j)} = \mathrm{ifft}(\hat{\mu}^{(j)}) + c_j$
    $\mathbf{h}^{(j)} \sim \max(0, \mu^{(j)} + \mathcal{N}(0, \mathrm{sigm}(\mu^{(j)})))$
    $\hat{\mathbf{h}}^{(j)} = \mathrm{fft}(\mathbf{h}^{(j)})$
    for $i = 1$ **to** N_c **do**
      $\Delta\hat{\mathbf{w}}^{(ij)} = \Delta\hat{\mathbf{w}}^{(ij)} + \overline{\hat{\mathbf{h}}^{(j)}} \cdot \hat{\mathbf{v}}^{(i)}$
      $\hat{\mathbf{v}}'^{(i)} = \hat{\mathbf{v}}'^{(i)} + \hat{\mathbf{w}}^{(ij)} \cdot \hat{\mathbf{h}}^{(j)}$
  $\forall i\colon \hat{v}'^{(i)}_{0,0} = \hat{v}'^{(i)}_{0,0} + N_\mathrm{v}^2 b_i$
  for $j = 1$ **to** N_k **do**
    $\hat{\mu}'^{(j)} = \sum_{i=1}^{N_\mathrm{c}} \overline{\hat{\mathbf{w}}^{(ij)}} \cdot \hat{\mathbf{v}}'^{(i)}$
    $\mu'^{(j)} = \mathrm{ifft}(\hat{\mu}'^{(j)}) + c_j$
    $\mathbf{h}'^{(j)} \sim \max(0, \mu'^{(j)} + \mathcal{N}(0, \mathrm{sigm}(\mu^{(j)})))$
    $\hat{\mathbf{h}}'^{(j)} = \mathrm{fft}(\mathbf{h}'^{(j)})$
    for $i = 1$ **to** N_c **do**
      $\Delta\hat{\mathbf{w}}^{(ij)} = \Delta\hat{\mathbf{w}}^{(ij)} - \overline{\hat{\mathbf{h}}'^{(j)}} \cdot \hat{\mathbf{v}}'^{(i)}$

(b) Training in the frequency domain

Figure 1.1: Comparison of training algorithms of convRBMs in (a) the spatial and (b) the frequency domain. Training in the frequency domain replaces the $5N_\mathrm{k}N_\mathrm{C}$ convolutions required in the spatial domain with simple element-wise multiplications, while adding only $4N_\mathrm{k}$ Fourier transforms. The other operations are equivalent in both domains.

operations needed for training (multiplications and additions) can be readily mapped to the frequency domain because the Fourier transform is a linear operation. Another necessary operation is the flipping of a convolutional kernel, $\tilde{w}(u, v) = w(N_\mathrm{w} - u + 1, N_\mathrm{w} - v + 1)$. To calculate flipping efficiently, we reindex the weights of a convolutional kernel $w_{uv}^{(ij)}$ from $u, v \in [1, N_\mathrm{w}]$ to $u, v \in [-\lfloor N_\mathrm{w}/2 \rfloor, \lfloor (N_\mathrm{w} - 1)/2 \rfloor$, which simplifies the calculation of

a flipped kernel to $\tilde{w}(u, v) = w(-u, -v)$. This allows flipping of a kernel in the spatial domain to be expressed by the element-wise calculation of the complex conjugate in the frequency domain, which follows directly from the time-reversal property of the Fourier transform, if $h(x) = f(-x)$, then $\hat{h}(\xi) = \hat{f}(-\xi)$; and the reality condition, $\hat{f}(-\xi) = \overline{\hat{f}(\xi)}$, where $\hat{x} = \mathcal{F}(x)$ denotes x in the frequency domain. Using the aforementioned mappings, equations (1.1) to (1.4) can be rewritten as

$$\Delta\hat{\mathbf{w}}^{(ij)} = \hat{\mathbf{v}}^{(i)} \cdot \overline{\hat{\mathbf{h}}^{(j)}} - \hat{\mathbf{v}}'^{(i)} \cdot \overline{\hat{\mathbf{h}}'^{j}} \tag{1.5}$$

$$\mathbb{E}[\hat{v}_{xy}^{(i)} \mid \hat{\mathbf{h}}] = \begin{cases} \sum_{j=1}^{N_\mathrm{k}} \hat{w}_{xy}^{(ij)} \hat{h}_{xy}^{(j)} + N_\mathrm{v}^2 b_i & \text{for } x, y = 0 \\ \sum_{j=1}^{N_\mathrm{k}} \hat{w}_{xy}^{(ij)} \hat{h}_{xy}^{(j)} & \text{for } x, y \neq 0 \end{cases} \tag{1.6}$$

$$\hat{\mathbf{h}}^{(j)} \sim \mathcal{F}\left(\max(0, \mathcal{F}^{-1}(\hat{\mu}^{(j)}) + c_j + \mathcal{N}(0, \sigma^2))\right) \tag{1.7}$$

$$\hat{\mu}^{j} = \sum_{i=1}^{N_\mathrm{c}} \overline{\hat{\mathbf{w}}^{(ij)}} \cdot \hat{\mathbf{v}}^{(i)} \tag{1.8}$$

where $\sigma^2 = \mathrm{sigm}(\mathcal{F}^{-1}(\hat{\mu}^{(j)}) + c_j)$, $\mathcal{F}^{-1}$ denotes the inverse Fourier transform, and $\cdot$ denotes element-wise multiplication. The algorithm for approximating the gradient in the frequency domain is summarized in Figure 1.1b.

The only operations that cannot be directly mapped to the frequency domain are the calculation of the maximum function, the generation of Gaussian noise, and trimming of the filter kernels. To perform the first two operations, an image needs to be mapped to the spatial domain and back. However, these operations need only be calculated $2N_\mathrm{k}$ times per iteration and are therefore not a significant contributor to the total running time. Because filter kernels are padded to the input image size, the size of the learned filter kernels must be explicitly enforced by trimming. This is done by transferring the filter kernels to the

spatial domain, setting the values outside of the specified filter kernel size to zero, and then transforming the filter kernels back to the frequency domain. This procedure needs to be performed only once per mini-batch. Since the number of mini-batches is relatively small compared to the number of training images, trimming of the filter kernels also does not add significantly to the total running time of the training algorithm.

1.3.3 Mapping Strided Convolutions to Stride-1 Convolutions

Convolutions with a stride $s > 1$ can be expressed equivalently as convolutions with stride $s = 1$ by reorganizing the values of $\mathbf{v}^{(i)}$ and $\mathbf{w}^{(ij)}$ to $\mathbf{V}^{(i')}$ and $\mathbf{W}^{(i'j)}$ as illustrated in Figure 1.2. This reindexing scheme allows the energy function to be expressed in terms of conventional (stride-1) convolutions, which facilitates training in the frequency domain. The new indices of $V^{(i')}_{x'y'}$ and $W^{(i'j)}_{u'v'}$ can be calculated from the old indices of $v^{i}_{(xy)}$ and $w^{(ij)}_{uv}$ as follows:

$$x' = \lfloor (x-1)/s \rfloor + 1 \qquad u' = \lfloor (u-1)/s \rfloor + 1 \tag{1.9}$$

$$y' = \lfloor (y-1)/s \rfloor + 1 \qquad v' = \lfloor (v-1)/s \rfloor + 1 \tag{1.10}$$

$$i' = s^2(i-1) + s((y-1) \bmod s) + ((x-1) \bmod s) + 1 \tag{1.11}$$

After reorganizing $\mathbf{v}^{(i)}$ and $\mathbf{w}^{(ij)}$ to $\mathbf{V}^{(i')}$ and $\mathbf{W}^{(i'j)}$, the energy of the model can be rewritten as

$$E(\mathbf{V}, \mathbf{h}) = -\sum_{i=1}^{N_\mathrm{C}} \sum_{j=1}^{N_\mathrm{k}} \sum_{x,y=1}^{N_\mathrm{h}} \sum_{u,v=1}^{N_\mathrm{W}} h^{(j)}_{xy} W^{(ij)}_{uv} V^{(i)}_{x+u-1,y+v-1} - \sum_{i=1}^{N_\mathrm{C}} b_i \sum_{x,y=1}^{N_\mathrm{V}} V^{(i)}_{xy} - \sum_{j=1}^{N_\mathrm{k}} c_j \sum_{x,y=1}^{N_\mathrm{h}} h^{(j)}_{xy} \tag{1.12}$$

$$= -\sum_{i=1}^{N_\mathrm{C}} \sum_{j=1}^{N_\mathrm{k}} \mathbf{h}^{(j)} \bullet (\tilde{\mathbf{W}}^{(ij)} * \mathbf{V}^{(i)}) - \sum_{i=1}^{N_\mathrm{C}} b_i \sum_{x,y=1}^{N_\mathrm{V}} V^{(i)}_{xy} - \sum_{j=1}^{N_\mathrm{k}} c_j \sum_{x,y=1}^{N_\mathrm{h}} h^{(j)}_{xy} \tag{1.13}$$

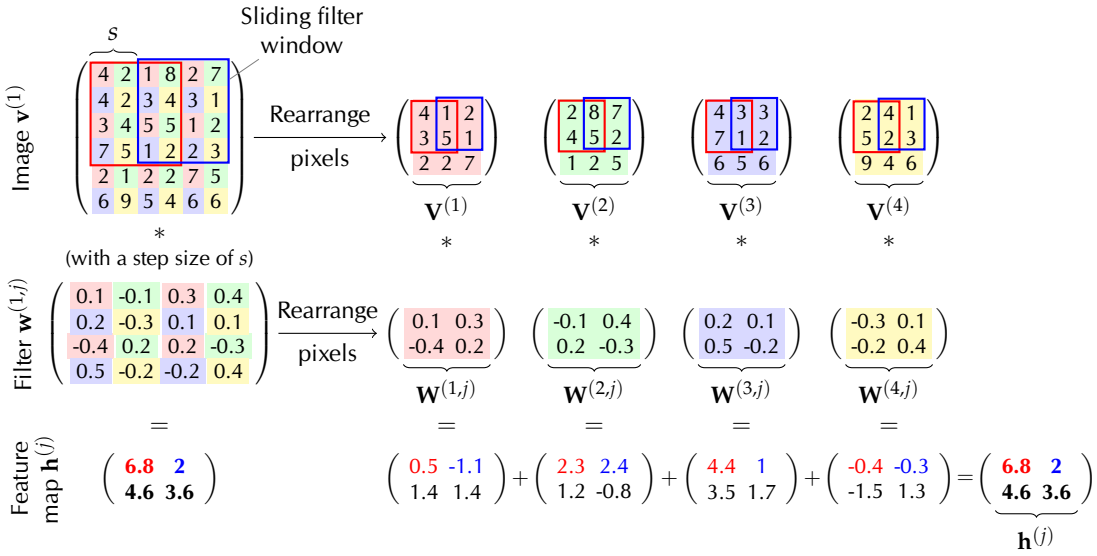


Figure 1.2: Illustration of convolutions with a sliding window step size $s = 2$ as used during filtering in a sconvDBN. A convolution of a given stride size (left side) can be efficiently calculated as the sum of multiple individual convolutions with $s = 1$ (right side) after rearranging the pixels of the input image and the filter kernels. The bottom row shows the actual values produced by the convolutions, which are the features from the image extracted by the filter. The figure is best viewed in color.

where $*$ denotes periodic convolution. The number of channels, number of visible units per channel and number of weights per channel after reorganization are given by $N_C = N_c \times s^2$, $N_V^2 = N_v^2/s^2$ and $N_W^2 = N_w^2/s^2$, respectively.

1.3.4 GPU Implementation and Memory Considerations

To further reduce training times, the algorithm can be efficiently implemented on graphics cards, which can perform hundreds of operations in parallel. To optimize efficiency, it is crucial to maximize the utilization of the large number of available GPU cores, while minimizing the required amount of GPU memory. Because our algorithm requires only a relatively small number of FFT calculations per iteration, the computational bottleneck is

the calculation of the element-wise operations, which can be performed in parallel. Thus we distribute the processing of a single 2D image over $N_\mathrm{v}(\lfloor N_\mathrm{v}/2 \rfloor + 1) \times N_\mathrm{c}$ independent threads, with one thread per element in the frequency domain. The large number of parallel threads results in a high utilization of the GPU, even when each image of a mini-batch is processed sequentially, which we do in our method because this greatly reduces the amount of GPU memory required to store the visible and hidden units compared to processing batches of images in parallel.

Due to the relatively small amount of GPU memory compared to CPU memory, memory requirements are an important consideration when designing a GPU algorithm. However, the total amount of required memory is highly implementation-dependent (e.g. the use of temporary variables for storing intermediate results) and a comparison can only be done with common elements at the algorithmic level. Therefore, in the remainder of this section, we focus on the memory requirements of key variables such as the visible units, the hidden units, and the filters, which are required by all implementations. In our training algorithm, all key variables are stored in the frequency domain, where each element in the frequency domain is represented by a single-precision complex number. Due to the symmetry of the Fourier space, the number of elements that need to be stored is roughly half the number of elements in the spatial domain. Thus, the memory required for storing the visible and hidden units in the frequency domain is roughly the same as in the spatial domain. A potential drawback of training in the frequency domain is that the filters need to be padded to the size of the visible units before applying the Fourier transformation, which increases the memory required for storing the filters on the GPU.

The total amount of memory in bytes required for storing the visible units, hidden units, and padded filters is given by the sum of their respective terms

$$4N_{\mathrm{v}}^2 N_{\mathrm{c}} + \frac{4N_{\mathrm{v}}^2 N_{\mathrm{k}}}{s^2} + 4N_{\mathrm{v}}^2 N_{\mathrm{k}} N_{\mathrm{c}}. \tag{1.14}$$

As a comparison, the training method of **?** processes batches of images in order to fully utilize the GPU, which requires the storing of batches of visible and hidden units. The memory needed for storing the key variables is given by

$$4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{c}} + \frac{4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{k}}}{s^2} + 4N_{\mathrm{w}}^2 N_{\mathrm{k}} N_{\mathrm{c}}, \tag{1.15}$$

where N_{b} is the number of images per mini-batch. Depending on the choice of the batch size, this method requires more memory for storing the visible and hidden units, while requiring less memory for storing the filters. Alternatively, **?** proposed to speed up the training of CNNs by calculating convolutions between batches of images and filters using FFTs. The memory required for storing the visible units, hidden units, and filters using this method is given by

$$4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{c}} + 4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{k}} + 4N_{\mathrm{v}}^2 N_{\mathrm{k}} N_{\mathrm{c}} \tag{1.16}$$

Table 1.1 shows a comparison of the memory per GPU required for storing key variables when training a network used in previous work by **?**. For the first layer, a comparison with Mathieu et al.'s training method could not be performed, because that method does not support strided convolutions, which would significantly reduce the memory required for storing the hidden units. In all layers, the proposed approach compensates for the increased memory requirements for the filters by considering one image at a time rather

Table 1.1: Comparison of the memory required for storing key variables using different training methods: our method (Freq), Krizhevsky et al.'s spatial domain method (Spat), and Mathieu et al.'s method using batched FFTs (B-FFT). A comparison with Mathieu et al.'s method could not be made for the first layer, because that method does not support strided convolutions. In all layers, our method consumes less memory for storing the key variables than the other two methods.

Layer	N_b	N_v	N_c	N_w	N_k	s	Memory in MB		
							Freq	Spat	B-FFT
1	128	224	3	11	48	4	28.7	147.1	—
2	128	55	48	5	128	1	72.9	260.5	330.9
3	128	27	128	3	192	1	69.2	114.8	182.3
4	128	13	192	3	192	1	24.0	33.0	55.5
5	128	13	192	3	128	1	16.1	27.3	42.3

than a batch, and still outperforms batched learning in the spatial domain in terms of speed (see Section 1.4), despite not using batches.

1.4 Evaluation of Running Time

1.4.1 Comparison of Running Times for Training sconvDBNs

To demonstrate where the performance gains are produced, we trained a two-layer sconvDBN on 2D and 3D images using our frequency-domain method and the following methods that all compute convolutions on the GPU, but using different approaches: 1) our spatial domain implementation that convolves a single 2D or 3D image with a single 2D or 3D filter kernel at a time, 2) Krizhevsky's spatial domain convolution implementation (**?**), which is a widely used method (e.g., **???**) that calculates the convolution of batches of 2D images and 2D filter kernels in parallel (note that this method cannot be applied to 3D images, so it was only used for the 2D experiments), and 3) our implementation that calculates convolutions using FFTs, but without mapping the other operations that would

Table 1.2: Training parameters.

Parameter	ImageNet (2D)		OASIS (3D)	
	1st layer	2nd layer	1st layer	2nd layer
Filter size	5 to 52	5 to 13	3 to 36	3 to 9
Stride size	1, 2, 4	1	1, 2, 4	1
Number of channels	3	16, 32, 64	1	8, 16, 32
Number of filters	32	32	32	16
Image dimension	512^2	128^2	128^3	64^3
Number of images	256	256	100	100
Batch size	128	128	25	25

Table 1.3: Hardware specification of our test system for training sconvDBNs using different implementations for calculating convolutions.

Processor	Intel i7-3770 CPU @ 3.40 GHz
CPU Memory	8 GB
Graphics Card	NVIDIA GeForce GTX 660
GPU Cores	960 cores @ 1.03 GHz
GPU Memory	2 GB

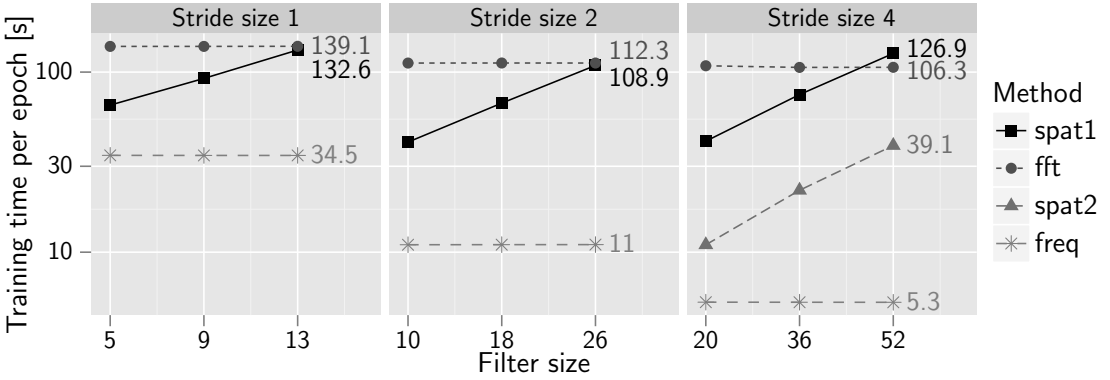
allow the algorithm to stay in the frequency domain when not computing convolutions. The parameters that we used for training convRBMs on 2D and 3D images are summarized in Table 1.2. The key parameters that we varied for our experiments are the filter size and stride size of the first layer, and the filter size and the number of channels of the second layer. Because the number of channels of the second layer is equal to the number of filters of the first layer, we also varied the number of filters of the first layer in order to attain the desired number of channels. For all implementations, the training time is directly proportional to the number of filters. Therefore, a detailed comparison of all four methods with a varying number of filters was not performed. The hardware details of our test environment are summarized in Table 1.3.

For the comparison on 2D images, we used a dataset of 256 natural color images from the ImageNet dataset (**?**). All images were resampled to a resolution of 512×512 pixels per color channel. For the evaluation on 3D images, we used 100 magnetic resonance images (MRIs) of the brain from the OASIS dataset (**?**). We resampled all volumes to a resolution of $128 \times 128 \times 128$ voxels and a voxel size of $2 \times 2 \times 2\,\text{mm}$.

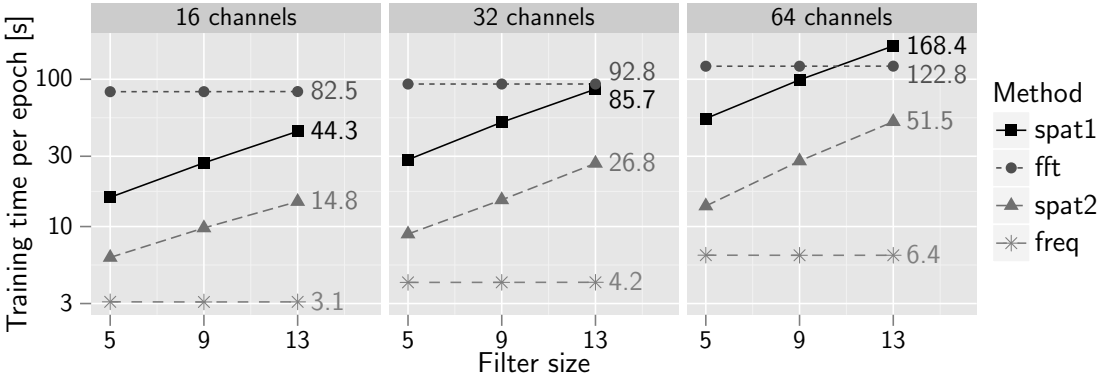
Running Time Analysis on 2D Color Images (ImageNet)

Figure 1.3(a) shows a comparison of running times for training the first sconvRBM layer on 256 images with varying filter and stride sizes. Due to internal limitations of Krizhevsky's convolution implementation, it cannot be applied to images with a resolution of 512×512 pixels when using a stride size smaller than four, and those comparisons could not be made. Our frequency domain implementation is between 2 to 24 times faster than our convolution implementation, where the speed gains are larger for larger filter and stride sizes. For a stride of one, the impact of the convolution implementation on the total running time is relatively low, because the computational bottleneck is the inference and sampling of the hidden units. As the number of hidden units decreases with larger strides, the running time becomes more dependent on the time spent to calculate convolutions. Hence, the differences between the four methods are more pronounced for larger strides. For a stride of four, training in the frequency domain is between 8 to 24 times faster than training in the spatial domain using our convolution implementation and 2 to 7 times faster than using batched convolutions. Calculating convolutions by FFTs is the slowest method for all stride sizes and 2D filter sizes up to 44, largely due to the cost of calculating Fourier transforms.

Figure 1.3(b) shows a similar comparison for training the second convRBM layer for a stride size of one and varying filter sizes and numbers of channels. In contrast to training



(a) Running times of training a first layer sconvRBM with stride sizes of 1, 2, and 4.



(b) Running times of training a second layer sconvRBM with 16, 32, and 64 channels (stride size 1).

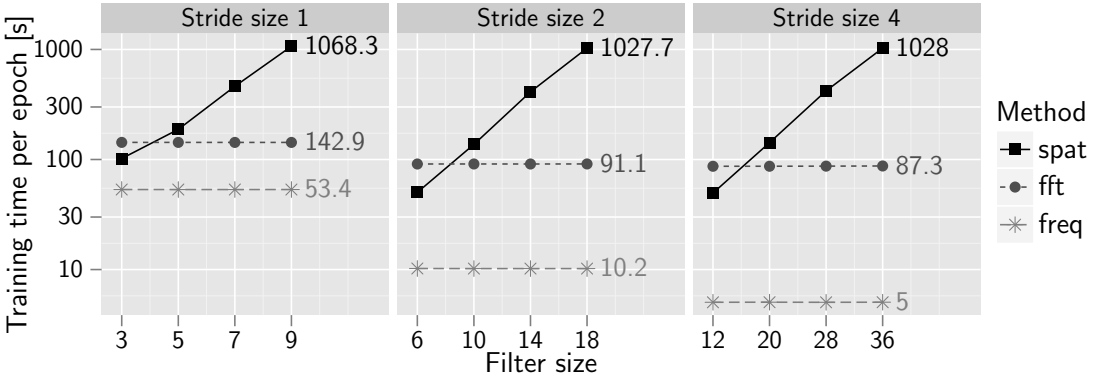
Figure 1.3: Comparison of running times for training a (a) first and (b) second layer sconvRBM on 2D images using our frequency domain method (freq) and three alternative methods using different convolution implementations: single image convolutions (spat1), batched convolutions (spat2), and convolution by using FFTs (fft). Due to internal limitations of the implementation of batched convolutions, a comparison with spat2 could not be performed for images with a resolution of 512×512 when using a stride size smaller than four.

the first layer, training times mostly depend on the calculation of convolutions, where the impact of calculating convolutions on the total running time increases with an increasing number of channels. Training in the frequency domain is between 5 to 26 times faster than training in the spatial domain using single-image convolutions, and 2 to 8 times faster than using batched convolutions. For all channel sizes, batched training is about 3 to 4 times

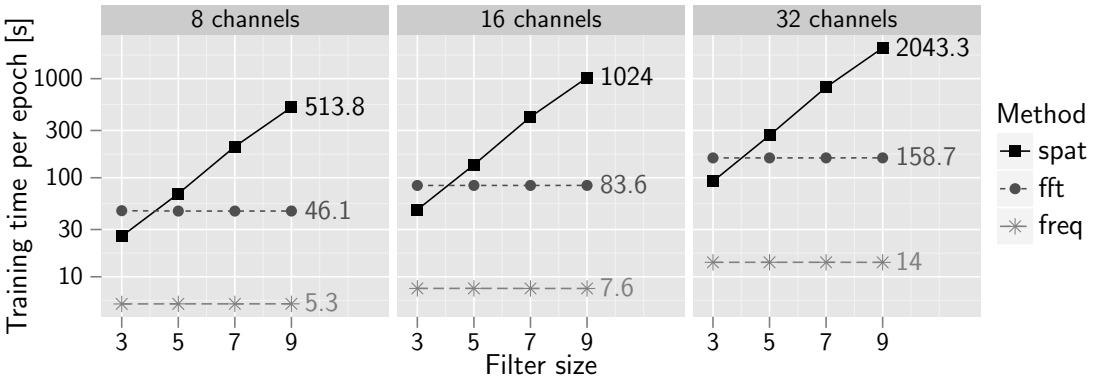
faster than non-batched training and calculating convolutions using FFTs is much slower than batched training and training in the frequency domain. To summarize, training of 2D images in the frequency domain is much faster than training in the spatial domain even for small filter sizes. Using the largest filter kernels in both layers, the proposed method is shown to yield a speedup of 7 to 8 times compared to state-of-the-art GPU implementations.

Running Time Analysis on 3D Volumes (OASIS)

Figure 1.4 shows the comparison of running times for training a first and second layer sconvRBM on 3D volumes for varying filter sizes, stride sizes, and varying numbers of channels. In contrast to training on 2D images, the computational costs of calculating 3D convolutions break even with calculating FFTs even for small filter sizes, because the number of multiplications and additions per convolution increases cubically, instead of quadratically, with the filter kernel size. As a result, simply training by convolutions in the frequency domain is faster than in the spatial domain. However, our proposed training algorithm still outperforms both other methods, even at the smallest filter size. For filter sizes of five and larger, our frequency domain implementation is between 3.5 to 200 times faster than our spatial domain implementation using single-image convolutions and 2.7 to 17 times faster than calculating convolutions by FFTs. Similar to the results on 2D images, training times of the first layer using a stride of one depend strongly on the time required to calculate the expectation of the hidden units and to sample the hidden units. Hence, performance improvements of our frequency domain method are more pronounced for larger strides and numbers of channels, where the impact of calculating convolutions on the total training time is also larger. This makes the proposed method particularly suitable for training sconvRBMs on high-resolution 3D volumes.



(a) Running times of training a first layer sconvRBM with stride sizes of 1, 2, and 4.



(b) Running times of training a second layer sconvRBM with 8, 16, and 32 channels (stride size 1).

Figure 1.4: Comparison of running times for training a (a) first and (b) second layer sconvRBM on 3D volumes using a single 3D image convolution implementation (spat), an implementation that calculates convolutions by using FFTs (fft), and our proposed implementation in the frequency domain (freq).

1.4.2 Comparison of Running Times of Selected Operations with cuDNN

The NVIDIA CUDA Deep Neural Network library (cuDNN) **?** is a GPU-accelerated library of primitives that are frequently used to implement deep learning methods such as CNNs and convDBNs. The library provides highly optimized implementations for calculating, e.g., batched 2D and 3D convolutions, pooling operations, and different transfer functions. In this set of experiments, we directly compared the time required to calculate convolutions

Table 1.4: CNN layer parameters for the comparison with cuDNN.

Experiment	#Channels	Image size	Filter size	Filter count	Batch size
Varying image size	2	$16^3, 17^3, \ldots, 128^3$	9^3	32	8
	32	$10^3, 11^3, \ldots, 64^3$	9^3	32	8
Varying batch size	2	128^3	9^3	32	1, 2, 4, 8
	32	64^3	9^3	32	1, 2, 4, 8
Varying filter size	2	128^3	$1^3, 2^3, \ldots, 9^3$	32	8
	32	64^3	$1^3, 2^3, \ldots, 9^3$	32	8

Table 1.5: Hardware specification of our test system for the comparison with cuDNN.

Processor	Intel i5-2500 CPU @ 3.30 GHz
CPU Memory	32 GB
Graphics Card	NVIDIA GeForce GTX 780
GPU Cores	2304 cores @ 0.94 GHz
GPU Memory	3 GB

in equations (??), (??), and (??) using our frequency domain implementation with cuDNN. We only consider convolutions because they are the most time-consuming operations and because all other operations are calculating in the same way. For the frequency domain implementation, all measurements include the time required to calculate FFTs for calculating gradients and for transforming the accumulated gradient to spatial domain in order to calculate the parameters updates. The parameters that we used for evaluating the speed of different convolution implementations are summarized in Table 1.4. The parameters represent typical values of the first and second layer of a CNN, which are the layers that typically take the most time to train. The key parameters that we varied are the image size, filter size, and batch size. The hardware details of our test environment are summarized in Table 1.5.

Figure 1.5 shows a comparison of the running times for calculating convolutions using our frequency domain implementation and cuDNN for varying image sizes. The FFT implementation that we used internally, cuFFT, is optimized for image sizes that can be factorized as $2^a \times 3^b \times 5^c$. Therefore, we also evaluated the time required to calculate convolutions when the input images are automatically padded to a size for which cuFFT has been optimized. For image sizes larger than 110^3, cuFFT requires significantly more temporary memory for calculating FFTs of unoptimized image sizes. For that reason, only the running times of the padded frequency domain implementation and cuDNN are shown for image sizes larger than 110^3. Calculating convolutions in the frequency domain scale better with increasing image size than cuDNN. While cuDNN is faster for very small image sizes, the padded frequency domain implementation is more than 20 times faster for images with a size of 128^3 voxels and 2 input channels, and more than 18 times faster for images with a size of 64^3 voxels and 32 channels. Calculating the FFT and therefore the running time of the frequency domain implementation varies significantly depending on the input image size, and careful padding of the input images is required to achieve consistently high performance.

Table 1.6 shows a comparison of running times per image for varying batch sizes and for varying number of channels. Increasing the batch size has only a minor effect on the running time of cuDNN, while significantly increasing the required amount of GPU memory. In contrast, the implementation in the frequency domain does not require more memory, because each image is processed individually. Increasing the batch size reduces the average running time per image of our frequency domain implementation, because the FFTs required to calculate the parameter updates need to be calculated only once per mini-batch, and have therefore a smaller impact on the overall running time, when the number of images per mini-batch is larger.

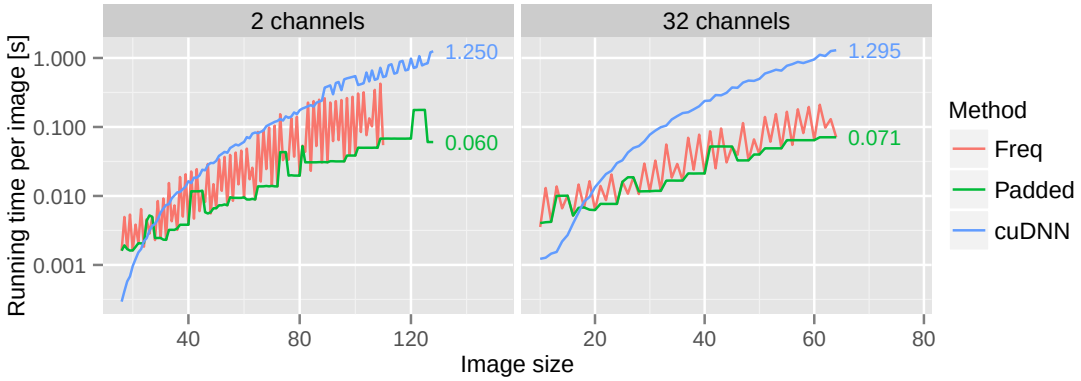


Figure 1.5: Comparison of running times of key operations for training a single CNN layer for varying number of channels and input sizes. Frequency domain training and training using cuDNN scales comparable for small numbers of channels. For large numbers of channels, frequency domain method scales slightly better to larger images.

- Impact of filter size on running time is shown in Figure 1.6.
- Surprisingly, training in the frequency domain is faster for all filter sizes, where the speed-ups are larger for larger filters.
- At a filter size of 9, the speed-up is 20 times.
- For 32 channels, cuDNN is faster for very small filter sizes and training in the frequency domain breaks even at a filter size of 3. For filter sizes larger than 3, training in the frequency domain is substantially faster with a speed-up of up to 18 times for a filter size of 9.

- Comparison of running time of critical operations for training the 7-layer CNN used for MS lesion segmentation
- Hidden layers benefit the most (10 times speed-up) from training in the frequency domain due to the large number of channels.

Table 1.6: Comparison of running times for calculating key operations for training a CNN layer for different batch sizes. Increasing the batch size reduces the impact of cropping the learned filters on the overall running time and consequently reduces the average time to process one image. The cuDNN implementation only benefits mildly from using larger batches.

Batch size	Running time [s]			
	2 channels		32 channels	
	Freq	cuDNN	Freq	cuDNN
1	0.109	1.397	0.153	1.313
2	0.081	1.011	0.106	1.307
4	0.068	1.249	0.082	1.304
8	0.060	1.247	0.071	1.296

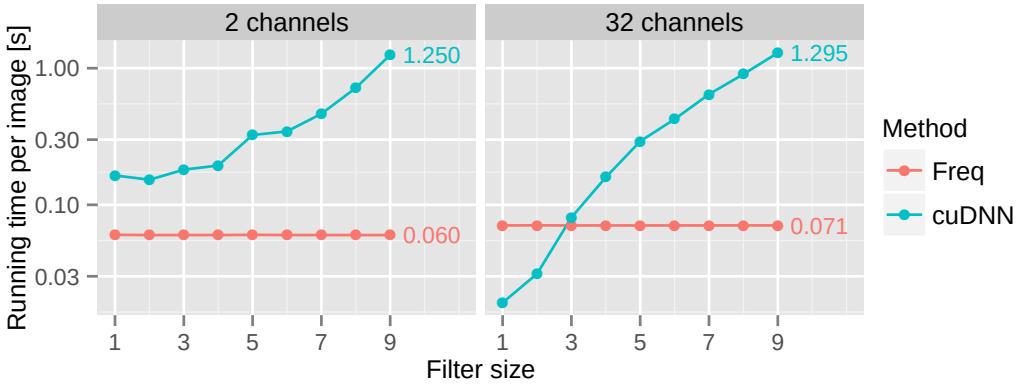


Figure 1.6: Comparison of running times of key operations for training a single CNN layer for varying number of channels and filter sizes.

- Due to the small number of input and output channels, the input and output layer benefit the least from training in the frequency domain
- Total speed-up compared to cuDNN is 6.3
- Comparison of running times of time critical operations for training the sconvRBM of the first layer of the lesion DBN
- Compared a direct calculation of strided convolution to remapping of strided convolutions to stride-1 convolutions.

Table 1.7: Comparison of running times of time critical operations of the 7-layer CEN-s used for segmentating lesions, and the first sconvRBM of the lesion DBN using to model lesion distribution. The running times of pooling layers we excluded.

Model		Running time [s]		Speed-up
		Freq	cuDNN	
7-layer CNN used for lesion segmentation	1st layer	0.077	0.498	6.507
	3rd layer	0.052	0.524	10.090
	4th layer	0.052	0.524	10.090
	6th layer	0.148	0.517	3.501
	Total	0.328	2.063	6.288
First sconvRBM of the lesion DBN	direct	0.076	0.068	0.904
	rearranged	0.019	0.071	3.786

- Strided convolutions using the reorganization trick is 4 times faster when calculated in the frequency domain.
- cuDNN does not benefit from reorganizing voxels
- Overall, training an sconvRBM in the frequency domain is 3.7 faster than using a direct implementation of strided convolutions from cuDNN.
- The architectures were not optimized for speed. Careful pre-padding of the images can be used to further reduce training times.