DEEP LEARNING FOR MANIFOLD LEARNING AND SEGMENTATION OF BRAIN MRIS

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)

November 2015

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.

Brosch, Tom:
Deep Learning for Manifold Learning and Segmentation of Brain MRIs
PhD thesis, The University of British Columbia, 2015

Contents

List of Tables

List of Figures

1 Training on Medical Images: Training in the Frequency Domain

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 (Hinton et al., 2006) 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. RainaEt al. (2009) 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, Lee et al. (2009, 2011) 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, Krizhevsky et al. (2012) 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 (Krizhevsky et al., 2012). An alternative approach was proposed by Mathieu et al. (2014) 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.

In this paper, we detail our training algorithm and GPU implementation in full, with a much more 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. Our 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., Mathieu et al., 2014), 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. We show that our method can be efficiently implemented on multiple graphics cards, further improving the runtime performance over other GPU-accelerated training methods. 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.

1.1 Algorithm

1.1.1 Training in the Spatial Domain

To train an sconvRBM on a set of images, the weights and bias terms 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 $\mathbf{V}_n, n \in [0, N-1]$ are reindexed 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

$$\mathbb{E}[\mathbf{V}^i \mid \mathbf{h}] = \textstyle\sum_j \mathbf{W}^{ij} * \mathbf{h}^j + b_i \tag{1.2}$$

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 (Nair and Hinton, 2010). The hidden units can be sampled with

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

$$\mu^j = \textstyle\sum_i \tilde{\mathbf{W}}^{ij} * \mathbf{V}^i + c_j \tag{1.4}$$

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

1.1.2 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 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 $\tilde{w}(a) = w(-a)$, which can be expressed by element-wise calculation of the complex conjugate; this follows directly from

input : Images $\mathcal{D} = \{\mathbf{V}_n \mid n \in [0, N-1]\}$
output: Weights gradient $\Delta\mathbf{W}$

$\Delta\mathbf{W} = 0$
foreach image $\mathbf{V} \in \mathcal{D}$ **do**
  $\mathbf{V}' = 0$
  for $j = 0$ **to** $N_\mathrm{k} - 1$ **do**
    $\mu^j = \sum_i \tilde{\mathbf{W}}^{ij} * \mathbf{V}^i + c_j$
    $h^j \sim \max(0, \mu^j + \mathcal{N}(0, \mathrm{sigm}(\mu^j)))$
    for $i = 0$ **to** $N_\mathrm{C} - 1$ **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 $j = 0$ **to** $N_\mathrm{k} - 1$ **do**
    $\mu'^j = \sum_i \tilde{\mathbf{W}}^{ij} * \mathbf{V}'^i + c_j$
    $\mathbf{h}'^j \sim \max(0, \mu'^j + \mathcal{N}(0, \mathrm{sigm}(\mu^j)))$
    for $i = 0$ **to** $N_\mathrm{C} - 1$ **do**
      $\Delta\mathbf{W}^{ij} = \Delta\mathbf{W}^{ij} - \tilde{\mathbf{h}}'^j * \mathbf{V}'^i$

(a) Training in the spatial domain

input : Images $\hat{\mathcal{D}} = \{\hat{\mathbf{V}}_n \mid n \in [0, N-1]\}$
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 $j = 0$ **to** $N_\mathrm{k} - 1$ **do**
    $\hat{\mu}^j = \sum_i \overline{\hat{\mathbf{W}}^{ij}} \cdot \hat{\mathbf{V}}^i$
    $\mu^j = \mathrm{ifft}(\hat{\mu}^j) + c_j$
    $\mathbf{h}^j \sim \max(0, \mu^j + \mathcal{N}(0, \mathrm{sigm}(\mu^j)))$
    $\hat{\mathbf{h}}^j = \mathrm{fft}(\mathbf{h}^j)$
    for $i = 0$ **to** $N_\mathrm{C} - 1$ **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 $j = 0$ **to** $N_\mathrm{k} - 1$ **do**
    $\hat{\mu}'^j = \sum_i \overline{\hat{\mathbf{W}}^{ij}} \cdot \hat{\mathbf{V}}'^i$
    $\mu'^j = \mathrm{ifft}(\hat{\mu}'^j) + c_j$
    $\mathbf{h}'^j \sim \max(0, \mu'^j + \mathcal{N}(0, \mathrm{sigm}(\mu^j)))$
    $\hat{\mathbf{h}}'^j = \mathrm{fft}(\mathbf{h}'^j)$
    for $i = 0$ **to** $N_\mathrm{C} - 1$ **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.

the time-reversal property of the Fourier transform and the reality condition $\hat{f}(-\xi) = \overline{\hat{f}(\xi)}$. Using the aforementioned mappings, equations ((1.1)) to ((1.4)) can be rewritten as

$$\Delta\hat{\mathbf{W}}^{ij} = \hat{\mathbf{V}}^i \cdot \overline{\hat{\mathbf{h}}^j} - \hat{\mathbf{V}}'^i \cdot \overline{\hat{\mathbf{h}}'^j} \tag{1.5}$$

$$\mathbb{E}[\hat{V}^i_{xy} \mid \hat{\mathbf{h}}] = \begin{cases} \sum_j \hat{W}^{ij}_{xy} \hat{h}^j_{xy} + N_\mathrm{V}^2 b_i & \text{for } x, y = 0 \\ \sum_j \hat{W}^{ij}_{xy} \hat{h}^j_{xy} & \text{for } x, y \neq 0 \end{cases} \tag{1.6}$$

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

$$\hat{\mu}^j = \sum_i \overline{\hat{\mathbf{W}}^{ij}} \cdot \hat{\mathbf{V}}^i \tag{1.8}$$

where $\sigma^2 = \mathrm{sigm}(\mathrm{F}^{-1}(\hat{\mu}^j) + c_j)$, $\hat{x} = \mathrm{F}(x)$ denotes x in the frequency domain, 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.

1.1.3 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

$$4N_{\mathrm{v}}^2 N_{\mathrm{c}} + \frac{4N_{\mathrm{v}}^2 N_{\mathrm{k}}}{s^2} + 4N_{\mathrm{v}}^2 N_{\mathrm{k}} N_{\mathrm{c}} \tag{1.9}$$

As a comparison, the training method of Krizhevsky et al. (2012) 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

$$4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{c}} + \frac{4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{k}}}{s^2} + 4N_{\mathrm{w}}^2 N_{\mathrm{k}} N_{\mathrm{c}} \tag{1.10}$$

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, Mathieu et al. (2014) 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

$$4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{c}} + 4N_{\mathrm{v}}^2 N_{\mathrm{b}} N_{\mathrm{k}} + 4N_{\mathrm{v}}^2 N_{\mathrm{k}} N_{\mathrm{c}} \tag{1.11}$$

Table 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 Krizhevsky et al. (2012). 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 than a batch, and still outperforms batched learning in the spatial domain in terms of speed (see Section 3), despite not using batches.

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

1.1.4 Mapping Strided Convolutional to Stride-1 Convolutions

Strided convolutions are a type of convolution that shifts the filter kernel as a sliding window with a step size or stride $s > 1$, stopping at only N_v/s positions. This reduces the number of hidden units per channel to $N_\mathrm{h} = N_\mathrm{v}/s$, hence significantly reducing training times and memory required for storing the hidden units during training. The energy of a strided convolutional RBM (sconvRBM) is defined as

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i=0}^{N_\mathrm{c}-1}\sum_{j=0}^{N_\mathrm{k}-1}\sum_{x,y=0}^{N_\mathrm{h}-1}\sum_{u,v=-\lfloor N_\mathrm{w}/2\rfloor}^{\lfloor (N_\mathrm{w}-1)/2\rfloor} h_{xy}^{j} w_{uv}^{ij} v_{sx+u,sy+v}^{i} - \sum_{i=0}^{N_\mathrm{c}-1} b_i \sum_{x,y=0}^{N_\mathrm{v}-1} v_{xy}^{i} - \sum_{j=0}^{N_\mathrm{k}-1} c_j \sum_{x,y=0}^{N_\mathrm{h}-1} h_{xy}^{j} \tag{1.12}$$

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 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

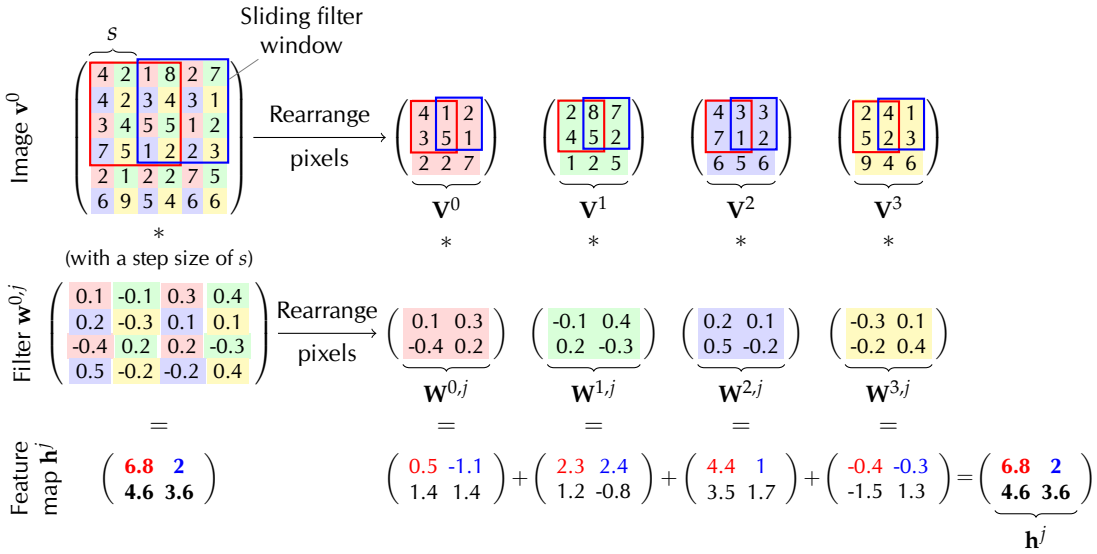


Figure 1.2: Illustration of convolutions with a sliding window step size $s = 2$ as used during filtering in a sconvDBN. A convolution of a given stride size (left side) can be efficiently calculated as the sum of multiple individual convolutions with $s = 1$ (right side) after rearranging the pixels 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.

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:

$$x' = \lfloor x/s \rfloor \qquad u' = \lfloor u/s \rfloor \tag{1.13}$$

$$y' = \lfloor y/s \rfloor \qquad v' = \lfloor v/s \rfloor \tag{1.14}$$

$$i' = s^2 i + s(y \bmod s) + (x \bmod s) \tag{1.15}$$

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

$$E(\mathbf{V},\mathbf{h}) = -\sum_{i=0}^{N_\mathrm{C}-1}\sum_{j=0}^{N_\mathrm{k}-1}\sum_{x,y=0}^{N_\mathrm{h}-1}\sum_{u,v=-\lfloor N_\mathrm{W}/2\rfloor}^{\lfloor (N_\mathrm{W}-1)/2\rfloor} h_{xy}^j W_{uv}^{ij} V_{x+u,y+v}^i - \sum_{i=0}^{N_\mathrm{C}-1} b_i \sum_{x,y=0}^{N_\mathrm{V}-1} V_{xy}^i - \sum_{j=0}^{N_\mathrm{k}-1} c_j \sum_{x,y=0}^{N_\mathrm{h}-1} h_{xy}^j \tag{1.16}$$

$$= -\sum_{i=0}^{N_\mathrm{C}-1}\sum_{j=0}^{N_\mathrm{k}-1} \mathbf{h}^j \bullet (\tilde{\mathbf{W}}^{ij} * \mathbf{V}^i) - \sum_{i=0}^{N_\mathrm{C}-1} b_i \sum_{x,y=0}^{N_\mathrm{V}-1} V_{xy}^i - \sum_{j=0}^{N_\mathrm{k}-1} c_j \sum_{x,y=0}^{N_\mathrm{h}-1} h_{xy}^j \tag{1.17}$$

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 $N_\mathrm{C} = N_\mathrm{c} * s^2$, $N_\mathrm{V}^2 = N_\mathrm{v}^2/s^2$ and $N_\mathrm{W}^2 = N_\mathrm{w}^2/s^2$, respectively.

1.2 Evaluation of Runtime

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 (Krizhevsky, 2012), which is a widely used method (e.g., Hinton and Srivastava, 2012; Scherer et al., 2010; Zeiler and Fergus, 2013) 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 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

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.

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

in Table 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 included in this paper. The hardware details of our test environment are summarized in Table Table 1.3.

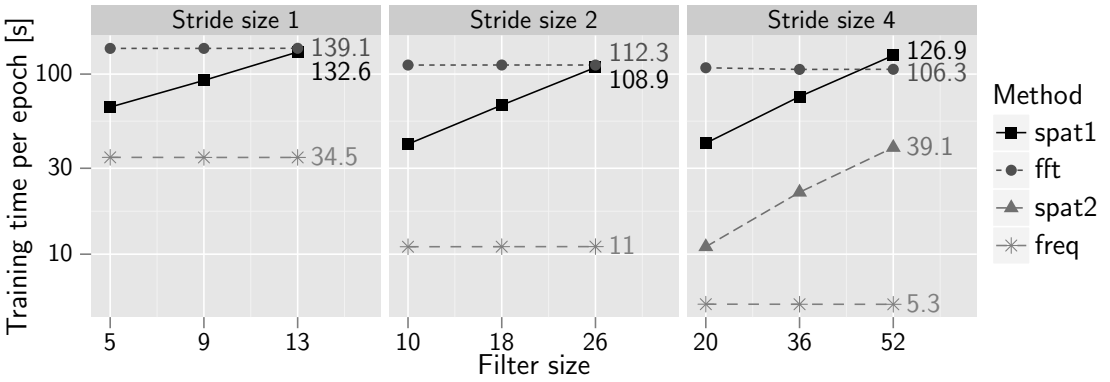
For the comparison on 2D images, we used a dataset of 256 natural color images from the ImageNet dataset (Deng et al., 2009). All images were resampled to a resolution of 512×512 pixels per color channel. For the evaluation on 3D images, we used 100

magnetic resonance images (MRIs) of the brain from the OASIS dataset (Marcus et al., 2007). We resampled all volumes to a resolution of $128 \times 128 \times 128$ voxels and a voxel size of $2 \times 2 \times 2\,\text{mm}$.

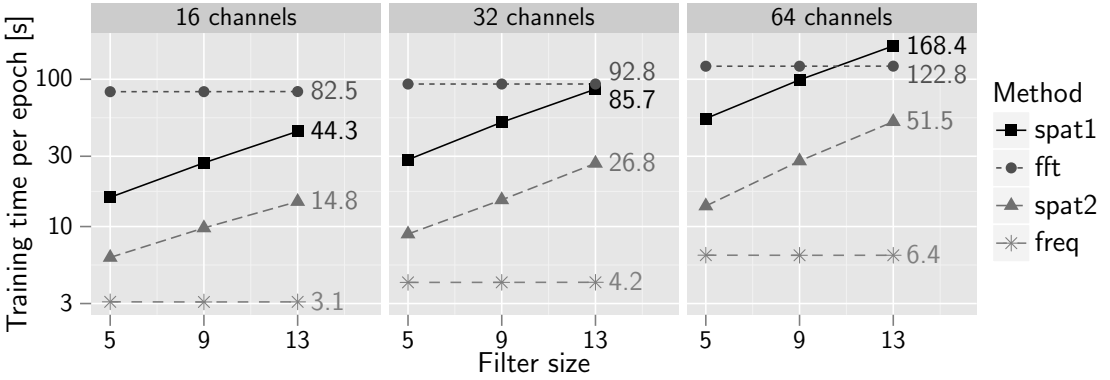
1.2.1 Running Time Analysis on 2D Color Images (ImageNet)

Figure 1.3a 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 1, 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 1.3b shows a similar comparison for training the second convRBM layer for a stride size of 1 and varying filter sizes and numbers of channels. In contrast to training 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



(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).

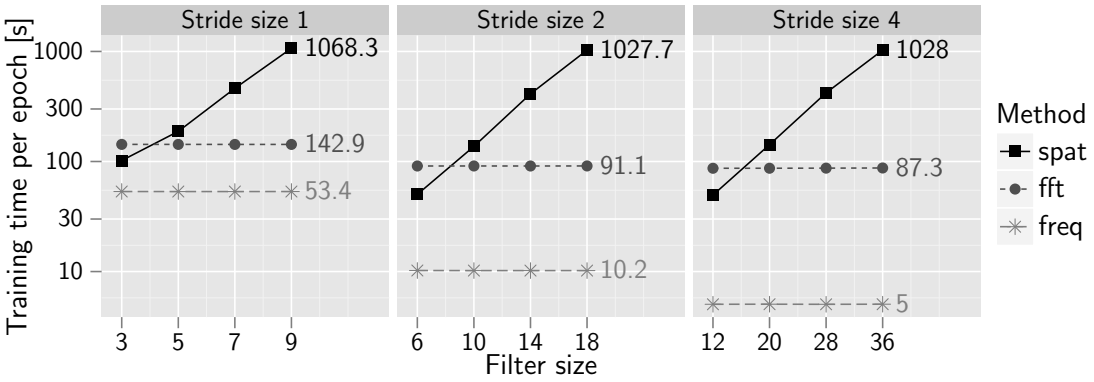
Figure 1.3: Comparison of running times for training a (a) first and (b) second layer sconvRBM on 2D images using our frequency domain method (freq) and three alternative methods using different convolution implementations: single image convolutions (spat1), batched convolutions (spat2), and convolution 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.

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

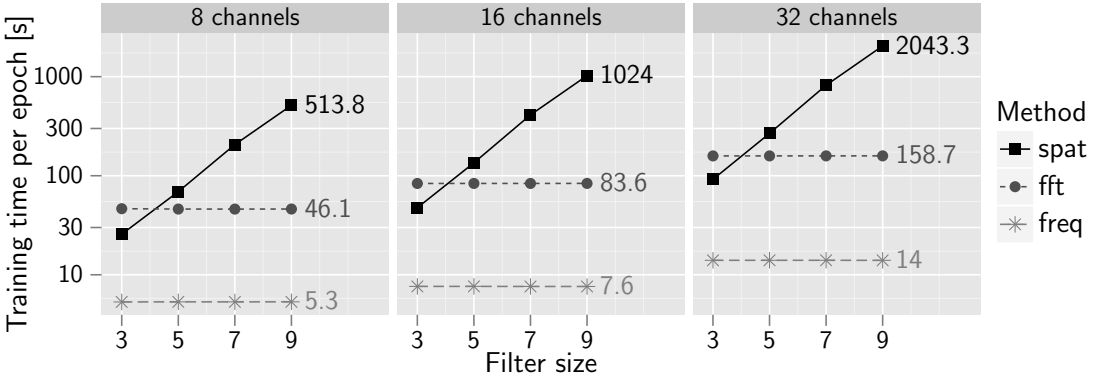
2D images in the frequency domain is much faster than training in the spatial domain even for small filter sizes. Using the largest filter kernels in both layers, the proposed method is shown to yield a speedup of 7 to 8 times compared to state-of-the-art GPU implementations.

1.2.2 Running Time Analysis on 3D Volumes (OASIS)

Figure 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 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 5 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 1 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).

1.2.3 Running Time Comparison with cuDNN on 3D Volumes

- Comparison of the runtime of time critical operations
- Only time required to calculate the convolutions in the forward pass, backward pass, and calculating the gradient of the filter kernels considered, since all other operations are the same

- For the frequency domain measurement, also the time to calculate the FFTs in order to trim the filter kernels was added.
- Varied parameters are the image size, the batch size and the filter size.
- All experiments were performed with only 2 layers and comparable large images, which represent a typical configuration of the first layer of deep learning model, and with medium sized images and 32 channels, which represents the second computational layer of a deep learning model.
- All parameter of the experiments are summarized in Table ??.
- Comparison of training on CNN layer using 3 different implementation: a) training in the frequency domain, b) training in the frequency domain after padding the input size to the next number $2^a \times 3^b \times 5^c \times 7^d$, and c) an implementation based on cuDNN.
- All methods require more time with increased image size, where the average running time over multiple sizes increases more for the cuDNN implementation.
- Calculating the FFT and therefore the running time of the frequency domain implementation varies significantly depending on a suitable input image size
- Padding is required to achieve consistently high performance compared to cuDNN
- Improvements over cuDNN are larger for higher number of channels where the impact of calculating the FFT on the overall running time is also lower
- Going from 2 to 32 channels, the observed speed-up increases from approximately 10 times for the 2 channel comparison to approximately 18 times for the 32 channel comparison.

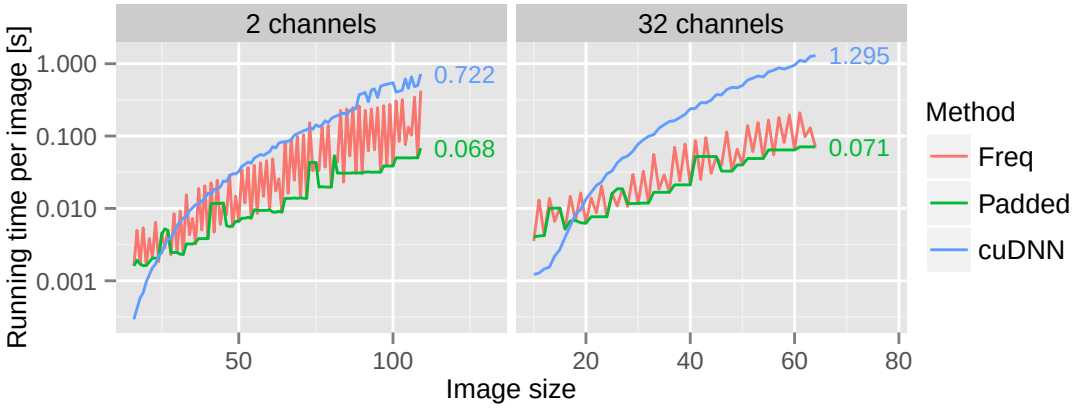


Figure 1.5: Comparison of running times of key operations for training a single CNN layer for varying number of channels and input sizes. Frequency domain training and training using cuDNN scales comparable for small numbers of channels. For large numbers of channels, frequency domain method scales slightly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

- Although the unpadded frequency domain implementation varies less for 32 channels than for the 2 channel case, padding is still advantageous.

- Table shows a comparison of running times 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, an implementation in the frequency domain does not require more memory, because each image is processed individually.

- Restricting the size of the learned filters, which requires additional calculations of the FFT, only need to be performed once per mini-batch

- Consequently, increasing the mini-batch size reduces the impact of filter truncating and therefore speeds-up the training

Table 1.4: 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

- 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.
- Due to the small number of input and output channels, the input and output layer benefit the least from training in the frequency domain

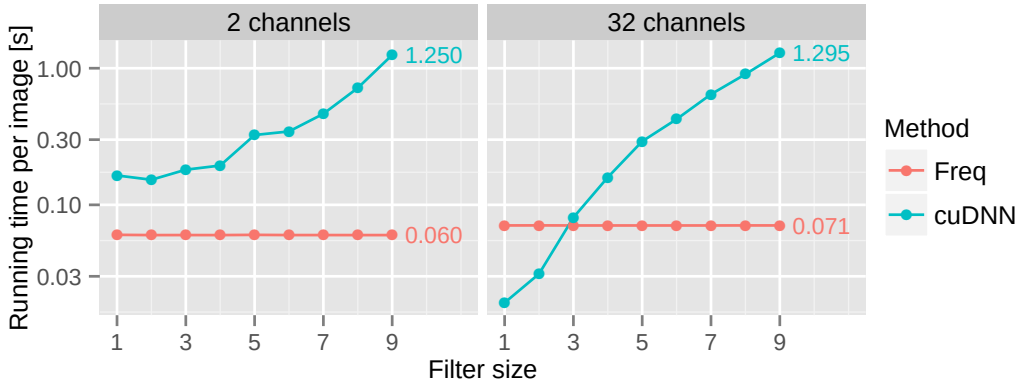


Figure 1.6: Comparison of running times of key operations for training a single CNN layer for varying number of channels and filter sizes.

- 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.
- 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.

Table 1.5: 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

Bibliography

Deng, Jia, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei (2009). *ImageNet: A large-scale hierarchical image database*. In *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. (cited on page 12)

Hinton, Geoffrey E., S. Osindero and Y.-W. Teh (2006). *A fast learning algorithm for deep belief nets*. Neural Computation, 18(7):1527–1554. (cited on page 1)

Hinton, Geoffrey E. and N. Srivastava (2012). *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv preprint arXiv:1207.0580, pp. 1–18. (cited on page 11)

Krizhevsky, Alex (2012). *High-performance C++/CUDA implementation of convolutional neural networks*. `http://code.google.com/p/cuda-convnet/`. (cited on page 11)

Krizhevsky, Alex, I. Sutskever and G. Hinton (2012). *ImageNet classification with deep convolutional neural networks*. In *Advances in Neural Information Processing Systems*, pp. 1–9. (cited on pages 2 and 8)

Lee, Honglak, R. Grosse, R. Ranganath and A. Y. Ng (2009). *Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations*. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 609–616. ACM. (cited on page 1)

Lee, Honglak, R. Grosse, R. Ranganath and A. Y. Ng (2011). *Unsupervised learning of hierarchical representations with convolutional deep belief networks*. Communications of the ACM, 54(10):95–103. (cited on page 1)

Marcus, Daniel S, T. H. Wang, J. Parker, J. G. Csernansky, J. C. Morris and R. L. Buckner (2007). *Open Access Series of Imaging Studies (OASIS): cross-sectional MRI data in young, middle aged, nondemented, and demented older adults.*. Journal of Cognitive Neuroscience, 19(9):1498–507. (cited on page 13)

Mathieu, Michael, M. Henaff and Y. LeCun (2014). *Fast Training of Convolutional Networks through FFTs*. In *2nd International Conference on Learning Representations*, pp. 1–9. (cited on pages 2 and 8)

Nair, Vinod and G. E. Hinton (2010). *Rectified linear units improve restricted Boltzmann machines*. In *Proceedings of the 27th Annual International Conference on Machine Learning*, pp. 807–814. (cited on page 4)

Raina, Rajat, A. Madhavan and A. Y. Ng (2009). *Large-scale deep unsupervised learning using graphics processors*. In *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880. ACM. (cited on page 1)

Scherer, Dominik, A. Müller and S. Behnke (2010). *Evaluation of pooling operations in convolutional architectures for object recognition*. In *International Conference on Artificial Neural Networks*, pp. 92–101. Springer. (cited on page 11)

Zeiler, Matthew D. and R. Fergus (2013). *Stochastic pooling for regularization of deep convolutional neural networks*. In *1st International Conference on Learning Representations*, pp. 1–9. (cited on page 11)