On CPU Performance Optimization of Restricted Boltzmann Machine and Convolutional RBM

Conference Paper · September 2016

DOI: 10.1007/978-3-319-46182-3_14

3 authors, including:

Baptiste Wicht
Université de Fribourg
15 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Jean Hennebert
University of Applied Sciences and Arts Western Switzerland
139 PUBLICATIONS **1,549** CITATIONS
SEE PROFILE

On CPU performance optimization of Restricted Boltzmann Machine and Convolutional RBM

Baptiste Wicht, Andreas Fischer, and Jean Hennebert

University of Applied Science of Western Switzerland
University of Fribourg, Switzerland
baptiste.wicht@hefr.ch, andreas.fischer@unifr.ch, jean.hennebert@hefr.ch

Abstract. Although Graphics Processing Units (GPUs) seem to currently be the best platform to train machine learning models, most research laboratories are still only equipped with standard CPU systems. In this paper, we investigate multiple techniques to speedup the training of Restricted Boltzmann Machine (RBM) models and Convolutional RBM (CRBM) models on CPU with the Contrastive Divergence (CD) algorithm. Experimentally, we show that the proposed techniques can reduce the training time by up to 30 times for RBM and up to 12 times for CRBM, on a data set of handwritten digits.

1 Introduction

Although most of the recent research has shown that learning on Graphics Processing Units (GPUs) is generally more efficient than training on Central Processing Units (CPUs) [13, 14, 20], especially for Convolutional Neural Networks (CNNs) [7, 9, 16], GPUs are not accessible everywhere. Some researchers may not have access to them and some laboratories may not want to upgrade their CPU clusters to GPU clusters. Therefore, it remains important to be able to train neural networks in reasonable time on machines equipped only with CPUs.

Restricted Boltzmann Machines (RBMs) are old models [19], that resurged recently to initialize the weights of an Artificial Neural Network (ANN) [4] or to extract features from samples [2]. Later on, the model was extended with the Convolutional RBM (CRBM) [11]. Performance optimization of these models was investigated on GPUs only [8, 15].

In the present paper, we present several techniques to reduce the training time of RBM and CRBM models. Techniques such as CPU vectorization, usage of BLAS kernels and reduction of convolutions to other functions are explored. To evaluate the performance, several networks are trained on 60'000 images of handwritten digits from the MNIST data set.

The rest of this paper is organized as follows. The system setup for the experiments is presented in Section 2. Section 3 presents techniques to speed up an RBM while optimizations for CRBM are detailed in Section 4. Section 5 covers the training of a DBN. Finally, conclusions are drawn in Section 6.

2 System Setup

The experiments have been computed on a Gentoo Linux machine with 12Go of RAM running an Intel® Core™ i7-2600 with a frequency of 3.40GHz. The tests were written in C++ using our own Deep Learning Library (DLL)[1] and Expression Templates Library (ETL)[2] libraries. The programs were compiled with GNU Compiler Collection (GCC) 4.9. Vector operations are vectorized using AVX. Intel® Math Kernel Library (MKL) is used as the BLAS implementation.

The experiments are conducted on the MNIST data set [10]. It contains grayscale images of handwritten digits, normalized to a size of 28x28 pixels. Each experiment is done on the 60'000 training images for 5 epochs and the average time per epoch is used as the final result.

3 Restricted Boltzmann Machine

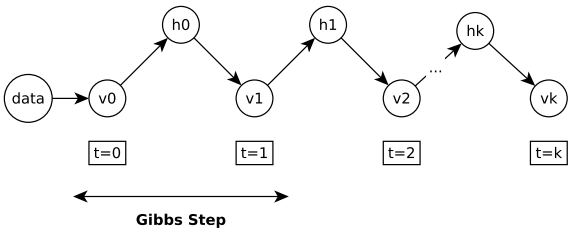

Fig. 1: Graphical representation of the Contrastive Divergence Algorithm. The algorithm CD-k stops at t=k. Each iteration performs a full Gibbs step.

A Restricted Boltzmann Machine (RBM) [19] is a generative stochastic Artificial Neural Network (ANN), developed to learn the probability distribution of some input. Training an RBM using the algorithm for general Boltzmann Machines [3] is very slow. Hinton et al. proposed a new technique, Contrastive Divergence (CD) [4], depicted in Figure 1. It is quite similar to the Stochastic Gradient Descent method, used to train regular ANNs. It approximates the Log-Likelihood gradients by minimizing the reconstruction error, thus training the model into an autoencoder. The algorithm performs a certain number of steps of Gibbs sampling (*CD-n*). When the RBM is used as a feature extractor or as a way of pretraining a Deep Belief Network [6], CD-1 is generally sufficient [5].

The original RBM model was designed with binary visible and binary hidden units (also called a Bernoulli RBM). Several different types of units were since developed (for instance Gaussian, ReLu or Softmax) [5]. This research focuses

[1] `https://github.com/wichtounet/dll/`
[2] `https://github.com/wichtounet/etl/`

on binary units, but the conclusions stand for all general types of units. Indeed, only the activation functions would change. The probability activations of visible and hidden units can be computed as follows:

$$p(h_j = 1|v) = \sigma(c_j + \sum_{i}^{m} v_i W_{i,j}) \tag{1}$$

$$p(v_i = 1|h) = \sigma(b_i + \sum_{j}^{n} h_j W_{i,j}) \tag{2}$$

The states of the units are obtained by sampling the activation probabilities. For binary units, Bernoulli sampling is performed to obtain the states:

$$s_j = \begin{cases} 1 & \text{if } p_j > \text{Unif}(0,1) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$s_i = \begin{cases} 1 & \text{if } p_i > \text{Unif}(0,1) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

From an implementation point of view, an RBM is made of a vector v of m visible units, a vector h of n hidden units, a matrix W of weights connecting the visible and the hidden units, a vector b of m visible biases and a vector c of n hidden biases. In practice, the weights are represented as single-precision floating point numbers rather than double-precision. Indeed, some single-precision computations can be as much as twice faster than their double-precision counterparts. Moreover, the precision is generally more than sufficient for CD training.

Algorithm 1 Standard CD-1 algorithm (one sample)

v_0 = training sample
h_0 = sample hidden activations from v_0
v_1 = sample visible activations from h_0
h_1 = sample hidden activations from v_1
$W_{pos} = v_0 \otimes h_0$
$W_{neg} = v_1 \otimes h_1$
$\nabla W = \epsilon(W_{pos} - W_{neg})$
$\nabla b = \epsilon(v_0 - v_1)$
$\nabla c = \epsilon(h_0 - h_1)$

Algorithm 1 describes the CD-1 algorithm for one sample. The same procedure is done for each sample of the data set and is repeated for as many epochs as necessary. In practice, it is important to note that the hidden activations should be computed directly from the visible activation probabilities rather than from the states [5]. Therefore, it is never necessary to compute the states of the visible

Table 1: Training time for an epoch of RBM training, in seconds. The speedup is the improvement gained by using BLAS kernels for linear algebra operations.

	A	B	C	D
Base	161.99	47.00	167.20	70.78
Base + BLAS	141.91	43.03	114.18	36.12
Speedup	1.14	1.09	1.46	1.95

units during training. Moreover, the last update of the hidden unit is only used to compute the positive gradients, in which case the probabilities are used rather than the states. Therefore, it is not necessary to sample the states of the hidden units for the last update.

In the algorithm and activation formulas, several computation routines are well-known and can be optimized. The Basic Linear Algebra Subprogams (BLAS) are a collection of small and highly optimized linear algebra routines. In the activation formulas, the sums are simply vector-matrix multiplication and matrix-vector multiplication. They can be implemented using the SGEMV operation from BLAS. The outer products to compute the positive and negative gradients can be implemented using the SGER routine. Finally, the computation of the visible and hidden biases gradients can be done with the SAXPY operation. For evaluation, the following networks are trained:

- `A`: 784 visible units, 500 hidden units
- `B`: 500 visible units, 500 hidden units
- `C`: 500 visible units, 2000 hidden units
- `D`: 2000 visible units, 10 hidden units

Table 1 shows the time, in seconds, necessary to train an epoch of the networks. Even if the computations are simple, BLAS operations can bring an important speedup to CD training, compared to standard implementations of these operations. For the tested networks, the speedup ranges from 1.09 to 1.95. The MKL BLAS implementation is highly tuned for Intel processor and each routine is especially optimized for cache and maximum throughput.

Experimentally, we find that more than 75% of the training time is spent inside the BLAS library, 8% in the sigmoid function and around 7% in random number generation. The sigmoid time could be optimized further by using an approximation of the sigmoid function or a vectorized version of the exponential function. Since this represents only a fraction of the total time, it would only slightly improve the general training time.

3.1 Mini-Batch training

In practice, CD is rarely performed one element at a time, but rather on a mini-batch. The data set is split into several mini-batches of the same size. The gradients are computed for a complete batch before the weights are updated. Algorithm 2 shows the updated version of CD-1 for mini-batch training.

Algorithm 2 Mini-batch CD-1 algorithm (one mini-batch)

for all $v_0 \in$ mini-batch **do**
  $h_0 =$ sample hidden activations from v_0
  $v_1 =$ sample visible activations from h_0
  $h_1 =$ sample hidden activations from v_1
  $W_{pos} \stackrel{+}{=} v_0 \otimes h_0$
  $W_{neg} \stackrel{+}{=} v_1 \otimes h_1$
end for
$\nabla W = \frac{\epsilon}{B}(W_{pos} - W_{neg})$
$\nabla b = \frac{\epsilon}{B}(v_0 - v_1)$
$\nabla c = \frac{\epsilon}{B}(h_0 - h_1)$

In practice, this could be implemented by accumulating the gradients element after element. However, it is better to compute the gradients independently for each element of the mini-batch. This needs more memory to store the intermediary results for the complete mini-batch. However, this is only for a small portion of the data set and has the advantage of allowing higher level optimizations of the loop body. Each iteration being completely independent, this could seem like an excellent candidate for parallelization. However, this is not the case. Depending on the dimensions of the matrices, a small speedup can be obtained by computing each iteration in parallel before aggregating the results sequentially. However, since most of the time will be spent in memory-bound operations (matrix-vector multiplication and outer product), there won't be enough bandwidth for many cores to process the data in parallel. A better optimization is to compute the activations and states of the units for a complete mini-batch at once instead of one sample at time. If we consider h as a $[B, n]$ matrix and v as a $[B, m]$ matrix, they can be computed directly as follows[3]:

$$h = \sigma(repmat(c, B) + v * W) \tag{5}$$

$$v = \sigma(repmat(b, B) + (W * h^T)^T) \tag{6}$$

This has the great advantage of performing a single large matrix-matrix multiplication instead of multiple small vector-matrix multiplication. In practice, this is much more efficient. In that case, the `SGEMM` operation of the BLAS library is used to compute the activation probabilities. Moreover, if the matrices are big enough, it is also possible to use a parallel version of the matrix-matrix multiplication algorithm. Figure 2 shows the time necessary to train each network with different batch sizes. It compares the base version with a hand-crafted matrix multiplication and the version using BLAS. The parallel BLAS version is also included in the results. On average the BLAS version is twice faster than the standard version and the parallel version of BLAS reduces the time by another factor of two.

[3] *repmat* vertically stacks the array B times

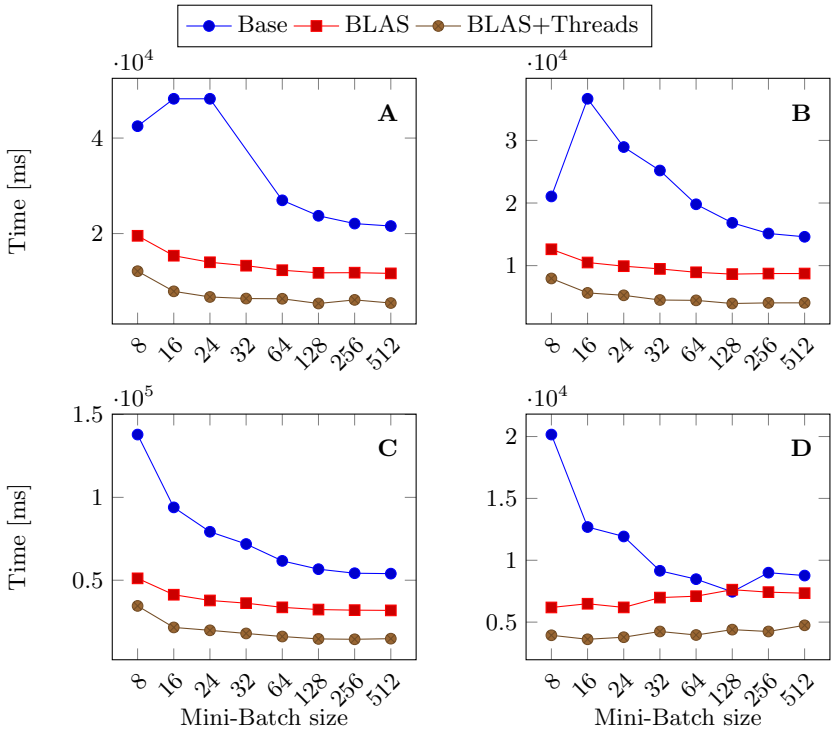

Fig. 2: Mini-Batch performance

Generally, increasing the mini-batch size reduces the training time. However, due to the small output dimension of the network `D`, the possible speedup is greatly reduced and larger mini-batch do not provide any substantial improvements. However, a too large mini-batch size may have negative impact on the classification performance of the network since many gradients will be averaged. On the other hand, a small batch size is generally leading to a more stable convergence. The batch size must be chosen as a trade-off between training time and classification performance. Moreover, a large mini-batch also increases the need for the inputs to be shuffled prior to each epoch. For MNIST, mini-batch size of up to 128 samples are still reasonable, but higher mini-batch are increasing the overall training time, by decreasing the learning each epoch does. To conclude this section, Table 2 compares the basic implementation and the final optimized version with mini-batch (128 samples) and a threaded BLAS library. Depending on the network, the optimized version is between 11 and 30 times faster.

Table 2: Final results for standard RBM training, in seconds.

	A	B	C	D
Base	161.99	47.00	167.20	70.78
Mini-Batch + BLAS + Threads	5.35	3.94	14.57	4.39
Speedup	30.27	11.92	11.47	16.12

4 Convolutional Restricted Boltzmann Machine

The original RBM model can be extended to the Convolutional Restricted Boltzmann Machine (CRBM) [11]. The visible and hidden layers are connected together by convolution, allowing the model to learn features shared among all locations of the input, thus improving translation invariance of the model. While this research focuses on two-dimensional CRBM, one-dimensional CRBM are also possible, for instance for audio [12] and the model can be adapted for three-dimensional inputs. Only square inputs and filters are described here for the sake of simplicity, but the model is able to handle rectangular inputs and filters.

A CRBM model has a matrix V of $C \times N_V \times N_V$ visible units. It has K groups of $N_H \times N_H$ hidden units. There are $C \times K$ convolutional filters of dimension $N_W \times N_W$ (by convolutional properties $N_W \triangleq N_V - N_H + 1$). There is a single visible bias c and a vector b of K hidden biases. The notation $\bullet_v$ is used to denote a `valid` convolution and $\bullet_f$ is for a `full` convolution. A tilde over a matrix ($\tilde{A}$) is used to indicate that the matrix is flipped horizontally and vertically. For a network with binary units, the probability activation are computed as follows:

$$P(h_{ij}^k = 1|v_c) = \sigma((\sum_c \tilde{W}_c^k \bullet_v v_c)_{ij} + b_k) \tag{7}$$

$$P(v_{cij}^k = 1|h) = \sigma((\sum_k W^k \bullet_f h^k)_{ij} + c) \tag{8}$$

A CRBM is trained similarly to an RBM, with an adapted version of formulas to compute the positive and negative gradients:

$$W_{ck}^{pos} = v_c^0 \bullet_v \tilde{h}_k^0 \tag{9}$$

$$W_{ck}^{neg} = v_c^1 \bullet_v \tilde{h}_k^1 \tag{10}$$

Training a CRBM requires a large number of convolutions for each epoch. Indeed, for each sample, there would be $2KC$ `valid` convolutions for the gradients, $2KC$ `valid` convolutions to compute the hidden activation probabilities (done twice in CD) and KC `full` convolutions for the visible units. Contrary to matrix multiplication, there is no general convolution reference implementation. The first optimization that can be applied is to vectorize the convolution implementations. Modern processors are able to process several floating point operations in one instruction. For instance, AVX instructions process 8 floats

at once, while SSE instructions process 4 floats once. While modern compilers are able to vectorize simple code, vectorizing complex program must be done by hand. We vectorized the inner loop of the convolutions, with AVX for large kernels and SSE for small kernels (smaller than 8 pixels). For evaluation, the following networks are trained:

- `A`: 1x28x28 visible units, 40 9x9 filters
- `B`: 40x20x20 visible units, 40 5x5 filters
- `C`: 40x16x16 visible units, 96 5x5 filters
- `D`: 96x12x12 visible units, 8 5x5 filters

Table 3 shows the time necessary to train the different networks and the obtained speedup. Due to the small images and filters in the measured networks, the speedups are only very interesting for the first layer of the network, with larger kernel and images. Moreover, the two-dimensional property of the algorithms adds overhead to the vectorized version, reducing the possible speedups.

Table 3: Results for Convolutional RBM training, in seconds.

	A	B	C	D
Base	380.37	3013.82	3947.46	338.16
Base + Vectorization	198.21	2174.66	3358.76	295.83
Speedup	1.91	1.38	1.17	1.14

When training the model using mini-batch, it becomes interesting to compute the gradients of each sample concurrently. For convolutions, there is no simple technique to compute the gradients of a complete batch, therefore parallelization inside batches is the best option. Figure 3 shows the results with different number of threads, with a mini-batch of 64. The performance increases almost linearly with the number of threads until four threads are used and then only slightly improves with more threads, exhibiting memory-bound computation behaviour. Since threads on the same core share the same cache, having more threads than cores does not improve the performance substantially in this case.

4.1 Valid convolution

As seen previously, training a CRBM requires four times more valid convolutions than full convolutions. Thus, it is extremely important to make it as fast as possible. By rearranging the image to be convolved, it is possible to reduce a valid convolution to a vector-matrix multiplication [18]. The general algorithm is presented in Algorithm 3.

However, because of the memory-inefficient im2col operation, this is experimentally slower than the vectorized version. Nevertheless, since the same image is convolved with K filters, the overhead of im2col can be greatly mitigated, by doing it only once for K convolutions. Moreover, the multiple vector-matrix

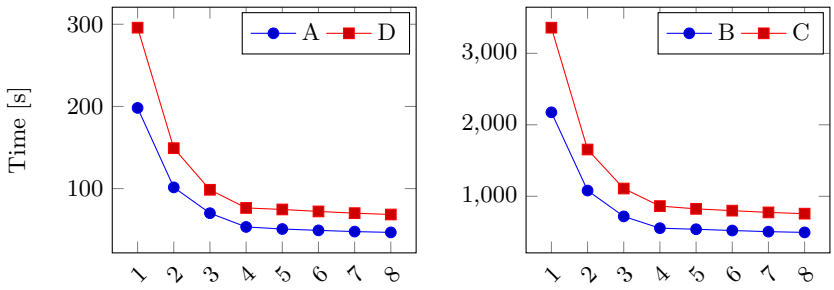

Fig. 3: Parallel performance

Algorithm 3 Convolution $C = I \bullet_v K$ with Matrix Multiplication

$W' = \text{reshape}(\tilde{W}, [1, k_1 k_2])$
$I' = \text{matrix}(k_1 k_2, c_1 c_2)$
$I' = \text{im2col}(K, [k_1 k_2])$
$C = W' * I'$

operations become a single matrix-matrix multiplication. Finally, since the computation of the activation probabilities and the gradients operates on flipped weights and that flipping is an involution, the computation can be done directly on the original weights, saving several flipping operations. Table 4 presents the results obtained when using this optimization for all the valid convolutions on the parallel version. On average, the training time is divided by two.

Experimentally, the difference in precision is found to be very small between the different versions and the reference. On average, the average difference between the vectorized version and the reference is in the order of $1e^{-5}\%$ and in the order of $5e^{-5}\%$ for the reduction with matrix multiplication. No difference has been observed when training a CRBM with different versions of the valid convolution. This difference may vary between different BLAS implementations.

Table 4: Results for Convolutional RBM training, in seconds.

	A	B	C	D
Parallel	46.69	494.52	756.70	68.47
Parallel + Reduction	28.45	241.79	336.56	40.12
Speedup	1.64	2.04	2.24	1.70

4.2 Full convolution

While there are no standard implementation of the full convolution, it is possible to reduce it to a another algorithm for which there exists efficient implementa-

tions. Using the convolution theorem, a full convolution can be reduced to a Fourier transform. Indeed, the convolution in the time domain is equal to the pointwise multiplication in the frequency domain [1]. Since the image and the kernel may not be of the same size, it is necessary to pad them with zeroes before computing their transforms. Algorithm 4 shows the steps used to compute a full convolution using a Fourier transform.

Algorithm 4 Convolution $C = I *_f K$ with Fourier Transform

$I' = \text{pad}(I)$
$K' = \text{pad}(K)$
$C' = \mathcal{F}(I') \cdot \mathcal{F}(I')$
$C = \mathcal{F}^{-1}(C')$

In practice, this can be implemented using the Fast Fourier Transform (FFT). There exists some standard and very efficient implementations of the FFT. While it is not a part of BLAS, the MKL library provides an FFT implementation.

Unfortunately, this is not always faster than a properly vectorized convolution. Table 5 shows the performance for different image and kernel sizes. The FFT convolution is around 3 times slower for an 16x16 image and a kernel of 5x5, while it is almost 12 times faster for an image of 256x256 and a kernel of 31x31. This shows that using an FFT algorithm to perform the full convolution can brings very large speedup to the training of a CRBM. However, it is only really interesting for large models. Another optimization that can be done when computing the full convolution by FFT is to precompute the Fourier transforms of the images. Indeed, each image will be convolved several times with different kernels, therefore only one transform per image is necessary. On the evaluated networks, this does not bring any substantial performance improvements. Only the network A has big enough images and kernels to profit from this, but this only result in a speedup of an epoch by less than 1%.

Again, the difference in precision is found to be very small. On average, the average difference for the vectorized version is found to be in the order of $1e^{-4}$% and in the order of $3e^{-5}$% for the FFT reduction. No difference has been observed when training a CRBM with different versions of the full convolution. The difference may vary between different FFT implementations.

Table 5: Performance of full convolution by FFT, in milliseconds

Image	12x12	16x16	16x16	28x28	50x50	128x128	128x128	256x256
Kernel	5x5	5x5	9x9	9x9	17x17	17x17	31x31	31x31
Vectorized	4.98	8.16	20.89	49.72	367.78	2010	7139	30787
FFT	11.59	24.49	25.8	46.38	122.43	368.83	1700	2598
Speedup	0.42	0.33	0.83	1.07	3.00	5.45	4.19	11.85

5 Deep Belief Network

A Deep Belief Network (DBN) is a network formed by stacking RBMs on top of each other. It is pretrained by training each layer with Contrastive Divergence. Once the first layer has been trained, its activation probabilities are computed for each input sample and these values are taken as the input of the next layer, and so on until the last layer of the network. A Convolutional DBN (CDBN) is similar, except that it stacks CRBMs.

Since pretraining a DBN consists in training RBMs with CD, the same optimizations discussed in previous sections apply. If there is enough memory, it is important to keep the entire data set in memory as well as the intermediate results (the activation probabilities of the previous layer) during training to maximize the performance. When this is not possible, the best course of action is to keep a multiple of the mini-batch size of samples in memory (and their intermediate output) for training. Ideally, computing the activation probabilities of the previous layer should be done in a separate thread so that CD always has data ready for training.

If the network is to be used for classification, it can then be fine-tuned using standard algorithms like for instance Stochastic Gradient Descent, Conjugate Gradient or Limited-Memory BFGS. Performance of these algorithms is out of the scope of this paper and has already been studied [17].

6 Conclusion and Future Work

Several techniques were presented to speedup training of RBM and CRBM models, on a single-CPU system. By using these techniques, RBM's training time has been reduced by up to 30 times and CRBM's training time has been reduced by up to 12 times. This demonstrates that even on CPU, many techniques can be used to substantially speedup the training of RBM models and train large models within reasonable time.

Future work could go in several directions. Combining several full convolutions together and using the FFT reduction could reduce its overhead and allow better performance even for small kernels. The performance of the vectorized convolution versions can also be improved further by vectorizing the convolution at the image level rather than just at the kernel level. Finally, once the large operations are fully optimized, operations such as sigmoid or Bernoulli sampling could also be considered for optimization.

References

1. Bracewell, R.: The Fourier transform and its applications. New York 5 (1965)
2. Coates, A., Ng, A.Y., Lee, H.: An analysis of single-layer networks in unsupervised feature learning. In: Proceedings of the Int. Conf. on Artificial Intelligence and Statistics. pp. 215–223 (2011)

3. Hinton, G.E., Sejnowski, T.J.: Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chap. Learning and Relearning in Boltzmann Machines, pp. 282–317. MIT Press, Cambridge, MA, USA (1986), http://dl.acm.org/citation.cfm?id=104279.104291
4. Hinton, G.E.: Training Products of Experts by minimizing Contrastive Divergence. Neural Computation 14, 1771–1800 (2002)
5. Hinton, G.E.: A practical guide to training Restricted Boltzmann Machines. In: Neural Networks: Tricks of the Trade, pp. 599–619. Springer (2012)
6. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science 313(5786), 504–507 (2006)
7. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia. pp. 675–678. ACM (2014)
8. Krizhevsky, A., Hinton, G.: Convolutional Deep belief Networks on CIFAR-10. Unpublished manuscript 40 (2010)
9. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
10. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)
11. Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional Deep Belief Networks for scalable unsupervised learning of hierarchical representations. In: Proceedings of the Int. Conf. on Machine Learning. pp. 609–616. ACM (2009)
12. Lee, H., Pham, P., Largman, Y., Ng, A.Y.: Unsupervised feature learning for audio classification using CDBNs. In: Proceedings of the Advances in Neural Information Processing Systems. pp. 1096–1104 (2009)
13. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al.: Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 451–460. ACM (2010)
14. Lopes, N., Ribeiro, B.: Towards adaptive learning with improved convergence of Deep Belief Networks on Graphics Processing Units. Pattern Recognition 47(1), 114–127 (2014)
15. Ly, D.L., Paprotski, V., Yen, D.: Neural networks on GPUs: Restricted Boltzmann Machines. see http://www.eecg.toronto.edu/~moshovos/CUDA08/doku.php (2008)
16. Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through ffts. arXiv preprint arXiv:1312.5851 (2013)
17. Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q.V., Ng, A.Y.: On optimization methods for deep learning. In: Proceedings of the 28th International Conference on Machine Learning (ICML-11). pp. 265–272 (2011)
18. Ren, J.S., Xu, L.: On vectorization of deep convolutional neural networks for vision tasks. arXiv preprint arXiv:1501.07338 (2015)
19. Smolensky, P.: Information processing in dynamical systems: Foundations of harmony theory. Parallel distributed Processing 1, 194–281 (1986)
20. Upadhyaya, S.R.: Parallel approaches to machine learning: A comprehensive survey. Journal of Parallel and Distributed Computing 73(3), 284–292 (2013)