GPU optimization of electroencephalogram analysis

Federico Raimondo[1], Juan Kamienkowski[2], and Diego Fernández Slezak[3]

[1] Instituto de Ciencias, Universidad Nacional de General Sarmiento
fraimondo@dc.uba.ar
[2] Departamento de Física, FCEyN, UBA, Pabellón 1, Ciudad Universitaria, (C1428EGA) Buenos Aires Argentina
juank@df.uba.ar
[3] Departamento de Física, FCEyN, UBA, Pabellón 1, Ciudad Universitaria, (C1428EGA) Buenos Aires Argentina
dfslezak@dc.uba.ar

Abstract. Nowadays, with the advent of new non-invasive techniques of brain imaging, researchers have access to neural processes underlying the cognition in humans. One of the main challenges in this techniques is the detection of patterns in brain signals, generally very noisy and with artifacts inserted by vital signs. One of the most successful techniques for this is Independent Component Analysis which detects statistically independent components that are produced from different sources. These methods are very expensive in computational time, with many hours of processing for a single experiment. We analyzed this algorithm and detect two main types of operations: vector-matrix and matrix-matrix. We implemented an ad-hoc solution that executes on GPU and compared this with the original and CUBLAS versions. We obtained a 4x and 40x of performance increase of vector-matrix and matrix-matrix operations, respectively. These results are the first step towards real-time EEG processing which may produce a significant advance into BCI applications.

1 Introduction

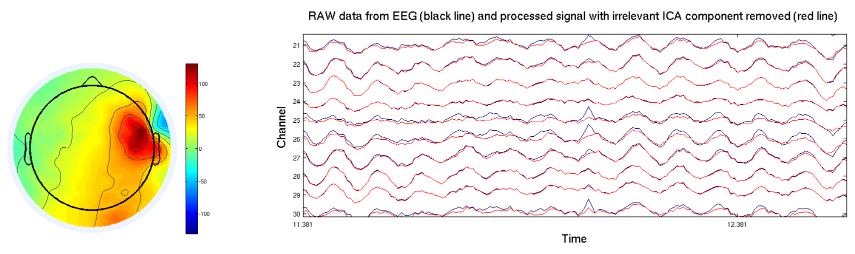
The new non-invasive brain imaging techniques provide access to neural processes underlying the cognition in human. However, all these methods involve two important limitations: (1) the high volume of data and (2) the signal and noise ratio and artifacts inserted by some vital signs. To attack the second, whether to eliminate the noise from signal or to identify the signal generated by these vital signs, a significant number of methods have been developed that require more and more resources that seriously compromises the task and becomes prohibitive for application in Brain Machine Interface.

A central problem in signal analysis is the search for a suitable representation or transformation to explain the observed signal. This multivariate statistical analysis for the separation of signals is a widely studied topic which is of great complexity because the sources have a lot of noise inherent in this kind of signals.

Many different approaches have been developed to separate the signals generated by the study of those sources that contribute only noise, such as PCA [13], factor analysis [8] and projection pursuit [7], among others.

In recent years, Independent Component Analysis (ICA) [3, 4] has come as an effective method for source separation and removal of noise and artifacts that proved to be useful in several scenarios. The most important are the separation of audio sources in noisy environments [4] and, in particular, brain imaging - electroencephalogram (EEG), magnetoencephalogram (MEG) and functional magnetic resonance imaging (fMRI) [14] - both for the removal of artifacts arising from eye movements [10] and for the analysis.

For example, the two most popular open packages for EEG analysis – EEGLAB (`http://sccn.ucsd.edu/eeglab/`) and Fieldtrip (`http://fieldtrip.fcdonders.nl/`) – make use of ICA strongly. In figure 1 we show an example of signals analyzed by ICA using EEGLAB. In the left panel, we show the scalp distribution of one independent component. In the right panel we show both raw and processed signals, observing that ICA extracts some unique features -as artifacts- included in the signal..



(a) Scalp distribution (b) RAW time series (black line) and the processed with artifacts removed by ICA (red line)

Fig. 1: Example of signals analyzed by ICA using EEGLAB.

As we pointed out, this method has two main challenges for widespread use into Brain-Computer Interface (BCI). A standard EEG experiment consists in the collection of data from 132 channels, each consisting of a time series at 512Hz of sampling rate. This data is stored in single-precision, which implies that a 3-hour experiment approximates a dataset size of 2.7GB, per subject. Moreover, analyzing this data with ICA requires a huge amount of computing power; for example, the 2.7GB dataset would take roughly 72 hours on a current desktop computer (Intel Core2Duo 4GB RAM), per subject using the implementations including in the standard toolboxes.

Current standard CPU hardware include no effective tools to operate with lots of floating point numbers. Today, these include advanced features such as Streaming SIMD Extensions (SSE) that allow this process to limited data simultaneously which makes them inefficient for this type of calculation. One approach to solving this problem is the use of Beowulf parallel computing clusters [11]. The main drawback of this implementation is the communication overhead that is needed to synchronize the different compute nodes involved.

We present a detailed study of operations involved in Infomax ICA [4] and implement GPU optimizations on relevant high-consumption functions for the decomposition of EEG signals and removal of artifacts.

2 Independent Component Analysis

The independent component analysis (ICA) is a concept introduced in 1994 by Pierre Common [6]. This analysis, of a random vector, consists of searching for a linear transformation that minimizes the statistic dependence between its components. In order to define suitable search criteria, the expansion of mutual information is utilized as a function of cumulants of increasing orders. The concept of ICA can be seen as an extension of the principal component analysis (PCA), which can only impose independence up to the second order, and consequently, defines directions that are orthogonal.

The following statistic model is assumed [9]:

$$x = My + v \tag{1}$$

where x, y and v are random vectors with values in $\mathbb{R}$ or $\mathbb{C}$ with zero mean and finite covariance, M is a rectangular matrix with at most as many columns as rows and vector y has statistically independent components. The problem set by ICA can be summarized as follows: given T samples of vector x, an estimation of matrix M is desired, and the corresponding samples from vector y. However, because of the presence of noise v, it is in general impossible to reconstruct the exact vector y. Since the noise v is assumed here to have an unknown distribution, it can only be treated as a nuisance, and the ICA cannot be devised for the noisy model above. Instead, it will be assumed that:

$$x = As \tag{2}$$

where s is a random vector whose components are maximizing a 'contrast' function. This constant vector s is maximum when its components are statistically independent.

Two variables y_1 and y_2 are independent if the information about the value of y_1 gives no information about y_2 and vice versa. Technically, independence can be defined by probability densities. Let $p(y_1, y_2)$ be the probability density function (pdf) of y_1 and y_2 and let $p_1(y_1)$ be the marginal pdf of y_1 then:

$$p_1(y_1) = \int p(y_1, y_2) dy_2 \tag{3}$$

y_1 and y_2 are independent if and only if the joint pdf can be factorized in the following way:

$$p(y_1, y_2) = p_1(y_1)p_2(y_2) \tag{4}$$

This definition applies for n variables, in which case the joint pdf must be a product of n terms.

This definition is used to derive into a more important property of random independent variables. Given two functions h_1 and h_2:

$$E\{h_1(y_1)h_2(y_2)\} = E\{h_1(y_1)\}E\{h_2(y_2)\} \tag{5}$$

where E is the expected value.

One fundamental restriction of ICA is that its independent components must be non-gaussian for ICA to be possible. To verify this restriction, the matrix A is assumed to be orthogonal and s_i gaussian. Then x_1 and x_2 are gaussian, uncorrelated and of unit variance. Then, the pdf, by definition, is:

$$p(x_1, x_2) = \frac{1}{2\pi} exp(-\frac{x_1^2 + x_2^2}{2}) \tag{6}$$

This density is completely symmetrical, therefore it does not contain any information on the directions of the columns of A. This is why A can not be estimated.

More rigorously, one can prove that the distribution of any orthogonal transformation of the gaussian (x_1, x_2) has exactly the same distribution as (x_1, x_2) and that x_1 and x_2 are independent. Thus, in the case of gaussian variables, we can only estimate the ICA model up to an orthogonal transformation.

In the original paper, the independence was measure by three different techniques (see [6] for the details):

- *Nongaussianity* as a measure of independence, estimaed by calculating the Kurtosis.
- *Negentropy*: tt is based on the information-theoretic quantity of differential entropy.
- Mutual Information Minimization.

2.1 Infomax (Information Maximization)

In [4], the authors presented a different approach to estimate the independent components. They proposed a neural network with three columns of neurons, each representing: (1) the original data (X); (2) the registered data (r); (3) the approximated independent data (Y). Each column of neurons combine linearly by matrixes A and W.

The principle used by this algorithm is maximizing the mutual information that output Y of a neural network processor contains about its input X. In this case, it is defined as

$$I(Y, X) = H(Y) - H(Y|X) \tag{7}$$

where $H(Y)$ is the entropy of output Y and $H(Y|X)$ is the entropy of the output that did not come from the input. In fact, $H(Y)$ is the differential entropy of Y with respect to some reference, such as the noise level or the accuracy of our discretization of the variables in X and Y. To solve this complexity, only the gradient of information-theoretic quantities with respect to some parameter w is considered. This gradients are as well behaved as discrete-variables entropies, because the reference terms involved in the definition of differential entropies disappear. The equation (7) can be differentiated, with respect to a parameter w as:

$$\frac{\partial}{\partial w} I(X,Y) = \frac{\partial}{\partial w} H(Y) \tag{8}$$

because $H(X|Y)$ does not depend on w.

In the system (1), $H(X|Y) = v$. Whatever the level of the additive noise, maximization of the mutual information is equivalent to the maximization of the output entropy, because $\frac{\partial}{\partial w} H(v) = 0$.

In consequence, for any invertible continuous deterministic mappings, the mutual information between inputs and outputs can be maximized by maximizing the entropy of the outputs alone.

For the unit case, let x be the input and $g(x)$ a transformation function so $g(x) = y$ with y the output, both $I(y,x)$ and $H(y)$ are maximized when the high density parts of the probability density function (pdf) of x are aligned with the high slopes of the function $g(x)$. This action can be described as "matching a neuron's input-output function to the expected distribution of signals" expressed in [12].

The natural (or relative) gradient method simplifies considerably the method. The natural gradient principle [1, 2] is based on the geometric structure of parameters space and it is related to the relative gradient principle [5] that ICA uses.

Using this approach, the authors propose the following iteration of the gradient method to estimate the W matrix:

$$\varDelta W \propto \; W - tanh(\frac{Wx}{2})(Wx)^T W \tag{9}$$

3 Implementation

In summary, ICA consists of the following steps. Data is represented as matrices, which must be operated in a number of preprocessing steps involving simple operations on them as centering around the mean, rotation, eigenvalues and eigenvectors. Current implementations (C/C++ and Matlab) make use of standard linear algebra libraries (BLAS) for the calculations involved.

1. $U = W \times perm(x)$ (where $perm$ is a random permutation)
2. $Y = -tanh(\frac{U}{2})$
3. $YU = Y \times U^T$

4. $YU = YU + I$
5. $W = lrate \times YU \times W + W$ ($lrate$ is the learning rate for each step)

As a first step, we performed a detailed analysis of operations involved in the method using *callgrind*, a tool for sequential profiling and optimization [15]. We executed the method for different datasets and profiled the function calls to the BLAS routines. In figure 2 we present an example of the call map of the Infomax ICA algorithm. We observe that BLAS routines `dgemv` and `dgemm` take more than 80% of total calculation time, so these are selected as candidates for GPU optimization.

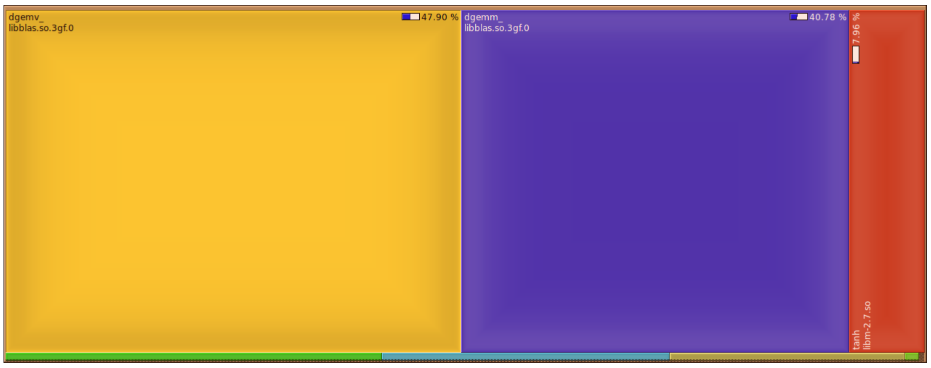


Fig. 2: Call map of Infomax ICA. BLAS routines dgemm and dgemv, which take around 80% of total time, are selected as candidates for GPU optimization.

As showed in figure 2 symbols `dgemv` and `dgemm` takes 47.9% and 40.78% of the total amount of time used to calculate ICA for a 136 channels dataset and 22528 samples. Recorded at 512 Hz, this dataset corresponds to a 44 seconds experiment. Using a 32 channels and 30504 samples dataset `dgemv` and `dgemm` takes 35.37% and 30.73% respectively. Further analysis on the C source code of the algorithm shows that both BLAS functions takes more percentage of time when the length of the dataset increases.

Real experiments durations are longer than 44 seconds. This time can scale up to hours, producing a 132 channels and millions of samples dataset where the BLAS functions optimizations would take almost 100% of time-usage.

`DGEMV` BLAS Matrix Vector multiplication

The symbol `dgemv` corresponds to BLAS function for matrix and vector multiplication.

```
void dgemv(char* trans, int m, int n, double alpha, double *a, int
lda, double *x, int incx, double beta, double * y, int incy)
```

This computes: `y = alpha * a * x + beta * y`

The corresponding parameters in Infomax calls are fixed so as to compute `y = a * x`, where a is a square matrix and the number of rows is equal to the number of channels in the dataset. The vector x corresponds to a single sample of number of channels dimension.

A naive and simple optimization can be done by using either CUDA or CUBLAS. For CUBLAS, just replacing the symbol `dgemv` for the corresponding symbol in the CUBLAS library:

```
void cublasDgemv (char trans, int m, int n, double alpha, const double
*A, int lda, const double *x, int incx, double beta, double *y, int
incy)
```

Nevertheless, both CUDA and CUBLAS uses a different memory space than standard sequential calculations: the video card RAM. Before starting computation, copying data from CPU memory to video memory must be performed.

The naive optimization for CUDA is as follows:

```
__global__ void matVec(double* wts, real* data, size_t wrowsize, size_t
    drowsize) {
    sharedsample[threadIdx.x] = data[blockIdx.x * drowsize + threadIdx.x];
    __syncthreads();

    double value = 0.0;
    for (int i = 0; i < blockDim.x; i++) {
        value += wts[threadIdx.x + i * wrowsize] + sharedsample[i];
    }

    data[threadIdx.x + drowsize * blockIdx.x] = value;
```

Taking advantage of CUDA shared memory space, this optimization first copies the vector into shared memory and then makes the matrix vector operation. This amount of shared memory space depends on CUDA version. Since version 2.0, CUDA supports double precision floating points operations and 48 KB of shared memory. This allows to compute matrix-vector operations for dimensions up to 6144 elements. Using aligned memory allows the usage of the optimum memory access resulting in 128 Byte transactions to main memory without discarding any data. Shared memory usage to copy the vector to be multiplied decreases the amount of global memory acceses by the amount of rows in the matrix minus one for each vector that is multiplied. The execution configuration syntax for this call to be done needs to set parameters correctly according to the dimension of both the matrix and vector.

`DGEMM` BLAS Matrix Matrix multiplication

The symbol `dgemm` corresponds to BLAS function for matrix-matrix multiplication.

```
dgemm(char *transa, char *trasnb, int m, int n, int k, double alpha,
double *a, int lda, double *b, int ldb, double beta, double *c, int
ldc)
```

This computes: `y = alpha * a * b + beta * c`

The corresponding parameters in Infomax calls are fixed so as to compute `c = a * b`, where the matrix `a` is squared and the number of rows is equal to the number of channels in the dataset.

In this case, the optimization for CUDA, is exactly the same as for `dgemv`. Again, for CUBLAS, symbol `dgemm` may be replaced for the corresponding symbol in the CUBLAS library:

```
void cublasDgemm (char transa, char transb, int m, int n, int k,
double alpha, const double *A, int lda, const double *B, int ldb, double
beta, double *C, int ldc)
```

4 Results

Tests have been made to compute matrix-vector and matrix-matrix multiplication on a Intel Core i7-2600 with 16 GB of RAM and a Nvidia Tesla C2070 video card. Performance comparisons were made using real datasets with the number of channels varying from 32 to 256 in 32 channel steps. The amount of samples vary from 15 minutes experiments to 105 minutes in 15 minutes steps, with a sampling rate of 512 Hz. More tests have been done with an Nvidia Quadro 4000 showing no significative difference regarding perfomance.

Table 1 shows the rate of performance increment in matrix-vector multiplication between BLAS and CUBLAS. This rate is calculated as $100 * timeOfCUBLAS/timeOfBLAS$. The matrix used is squared and has as many rows as channels in the dataset. The results show the percentage of time CUBLAS take in comparison to BLAS for the multiplication of the matrix by every sample in the dataset. Surprisingly, we observe more than 30x performance decrease with the use of CUBLAS routines on executions with little number of channels. This bad behavior resides on the fact of inefficient memory allocation: CUDA has very restrictive memory alignment directions in order to achieve maximum performance. CUBLAS makes no use of them. Further analisys using NVIDIA Compute Visual Profiler showed that two factors where the key to this performance decrease: global and shared memory usage, and active threads per processor. CUBLAS implementation doubled the amount of global memory acceses while made half the shared memory ones in comparison with CUDA ad-hoc implementation. The other factor is the parallel usage of processors in the GPU: CUBLAS kept an average of 4.24 active warps per cycle, while CUDA average was 34.73. As the number of channels grow, times for both implementations get closer, obtaining almost same performance in the biggest datasets. This results suggest that an ad-hoc solution as proposed may be necessary in order to take advantage of GPU architecture.

	Experiment length (minutes)						
Channels	15	30	45	60	75	90	105
32	3,387.18	3,441.56	3,472.81	3,431.82	3,495.81	3,493.01	3,443.28
64	1,096.45	1,097.52	1,087.50	1,088.67	1,089.22	1,091.38	1,100.20
96	600.68	599.83	601.81	599.83	598.91	597.96	599.42
128	392.63	394.22	394.35	394.27	393.11	395.85	394.77
160	285.94	284.78	286.16	285.22	287.14	287.45	284.85
192	219.71	218.57	218.50	219.82	219.23	219.09	218.05
224	178.45	177.23	176.70	178.08	177.20	177.94	
256	150.24	150.72	150.40	149.74	150.47		

Table 1: Rate of performance increment between BLAS and CUBLAS in matrix-vector multiplication. CUBLAS show a very inferior performance than BLAS, presumably because of bad memory alignment. This rate is calculated as $100 * timeOfCUBLAS/timeOfBLAS$.

Similarly, table 3 shows the results comparing the CUDA ad-hoc implementation and BLAS.

In this case of matrix-vector multiplication, we observe a 4x performance increment using the ad-hoc GPU specific implementation developed. These results are possible to obtain because the ad-hoc solution takes into account the data structures and GPU architecture, making a very efficient use of video card memory. On the other hand, CUBLAS is a general-purpose implementation that does not take into account specific characteristics of problem involved and may lead to inefficient memory operations.

Same executions have been made for matrix-matrix multiplication. Table 2 shows the comparison between the ad-hoc CUDA implementation and BLAS while table 3 show results for CUBLAS and the ad-hoc CUDA implementation.

As expected, in this case CUBLAS performs the best between the three implementations. This behavior is caused by the fact that ad-hoc CUDA implementation does not take into account several factors that maximizes the GPU processors throughput and more than half of the multiprocessors in the video card might be idle during the entire operation.

Nevertheless, our implementation showed a 4x performance increment to the original BLAS routine.

The out-of-the-shelf CUBLAS routine performed the best, obtaining increments of 10x to 20x in matrix-matrix multiplication performance compared to the ad-hoc developed one, as shown in table 3. This results show almost a 40x increment of performance totally using the CUBLAS for matrix-matrix multiplication rather than the standard BLAS.

Because of the nature of CUBLAS and the optimization made with CUDA for matrix-vector multiplication, it is not trivial to implement an hybrid solution

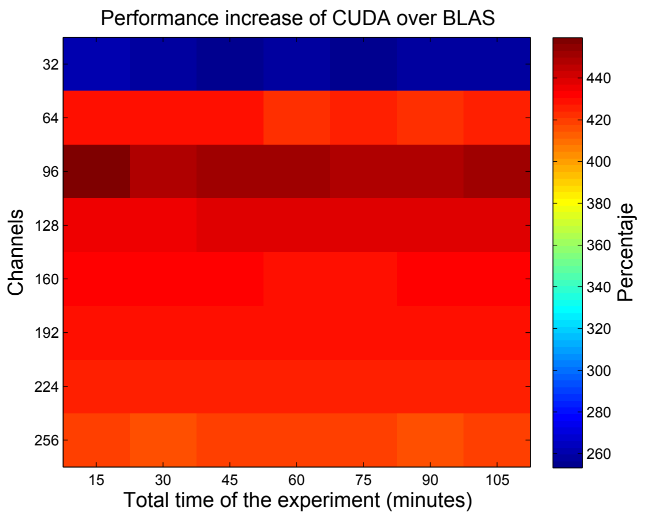


Fig. 3: Rate of performance increment between ad-hoc CUDA implementation and BLAS in matrix-vector multiplication. This rate is calculated as $100 * timeOfBLAS/timeOfCUDA$

	Experiment length (minutes)						
Channels	15	30	45	60	75	90	105
32	353.33	351.72	333.33	338.98	337.84	340.45	338.83
64	457.58	469.70	463.27	455.64	459.39	457.29	457.33
96	485.94	476.92	474.49	474.71	475.15	475.19	476.70
128	457.89	457.89	456.85	456.11	457.09		
160	443.58	445.79	446.07	445.58			
192	438.52	440.51	440.23				
224	435.06	435.06	435.82				
256	421.81	421.59					

Table 2: Rate of performance increment between the ad-hoc CUDA implementation and BLAS in matrix-matrix multiplication. This rate is calculated as $100 * timeOfBLAS/timeOfCUDA$

	Experiment length (minutes)						
Channels	15	30	45	60	75	90	105
32	1,500.00	1,450.00	1,500.00	1,475.00	1,480.00	1,483.33	542.11
64	3,300.00	2,200.00	2,450.00	2,216.67	2,357.14	2,487.50	2,320.00
96	1,280.00	1,300.00	675.86	745.71	815.00	651.67	700.00
128	2,280.00	2,072.73	2,143.75	2,180.95	2,196.15		
160	1,491.67	1,547.83	1,525.71	1,584.44			
192	2,336.36	2,221.74	2,194.29				
224	1,657.14	1,697.56	1,683.87				
256	2,161.90	2,214.63					

Table 3: Rate of performance increment between CUBLAS and Cuda in matrix-matrix multiplication. This rate is calculated as $100 * timeOfCUDA/timeOfCUBLAS$

merging both versions. Out matrix-vector operation relies on memory alignment and access conditions so matrixes are stored with padding between columns, while CUBLAS matrix-matrix operations needs a contiguous matrix in memory. Both approaches were tested: a CUBLAS-only version of Infomax ICA with a result of 50% increase in the amount of the time required to perform and a CUDA-only version with both ad-hoc optimizations showin a 4x increase in perfomance.

The matrix-vector optimization applied to Infomax ICA consisted on three main optimizations. Both optimizations on memory usage described previously and a new one: Combining asynchronous execution between CPU and GPU. Random permutations can be achieved by generating a random permutation vector of integers in the CPU while computing any other operation in GPU and using the generated integers as indexes in the data matrix to locate the corresponding randomly permuted vector.

5 Conclusions

Nowadays, with the advent of new non-invasive techniques of brain imaging, researchers have access to neural processes underlying the cognition in humans. ICA arises as a very useful method for brain signal analysis for source separation and removal of noise and artifacts. On the other hand, analyzing data with ICA requires a huge amount of computing power. We presented a detailed study of operations involved in Infomax ICA which showed that vector-matrix and matrix-matrix operations take almost all computational time. Based on these results, we implemented an ad-hoc solution for GPU optimizations. We compared the developed solution with the original BLAS and CUBLAS implementations. In this case of matrix-vector multiplication, we observed a 4x performance increment using the ad-hoc GPU specific implementation developed. CUBLAS behaved worse than the original one, caused by the inefficient use of memory. With matrix-matrix operations, CUBLAS routine performed the best, obtaining increments of 40x. These promising results suggest that the hybrid implementation of ICA using the ad-hoc solution for matrix-vector operations and CUBLAS for matrix-matrix cases would present a very significant performance increase in ICA calculation.

Acknowledgements

This work was supported by the Nvidia Academic Partnership.

References

1. Amari, S.: A new learning algorithm for blind signal separation. Advances in Neural Information Processing Systems (1996)
2. Amari, S.: Neural learning in structured parameter spaces-natural Riemannian gradient. In: In Advances in Neural Information Processing Systems. Citeseer (1997)
3. Amari, S.: Natural gradient works efficiently in learning. Neural computation 10(2), 251–276 (1998)
4. Bell, A., Sejnowski, T.: An information-maximization approach to blind separation and blind deconvolution. Neural computation 7(6), 1129–1159 (1995)
5. Cardoso, J., Laheld, B.: Equivariant adaptive source separation. Signal Processing, IEEE Transactions on 44(12), 3017–3030 (1996)
6. Comon, P.: Independent component analysis, a new concept? Signal processing 36(3), 287–314 (1994)
7. Friedman, J.: Exploratory projection pursuit. Journal of the American statistical association 82(397), 249–266 (1987)
8. Harman, H.: Modern factor analysis. University of Chicago Press (1976)
9. Hyvärinen, A., Oja, E.: Independent component analysis: algorithms and applications. Neural networks 13(4-5), 411–430 (2000)
10. Jung, T., Makeig, S., McKeown, M., Bell, A., Lee, T., Sejnowski, T.: Imaging brain dynamics using independent component analysis. Proceedings of the IEEE 89(7), 1107–1122 (2002)
11. Keith, D., Hoge, C., Frank, R., Malony, A.: Parallel ICA methods for EEG neuroimaging. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. p. 10. IEEE (2006)
12. Laughlin, S.: A simple coding procedure enhances a neuron's information capacity. Z. Naturforsch 36(9-10), 910–912 (1981)
13. Oja, E.: Principal components, minor components, and linear neural networks. Neural Networks 5(6), 927–935 (1992)
14. Schöpf, V., Kasess, C., Lanzenberger, R., Fischmeister, F., Windischberger, C., Moser, E.: Fully Exploratory Network ICA (FENICA) on resting-state fMRI data. Journal of Neuroscience Methods (2010)
15. Weidendorfer, J.: Sequential performance analysis with callgrind and kcachegrind. Tools for High Performance Computing pp. 93–113 (2008)