Neural Network Tool for Data Mining: SOM Toolbox

Juha Vesanto

SOM Toolbox Team
Helsinki University of Technology
Neural Networks Research Centre
P.O.Box 5400, FIN-02015 HUT, Finland
somtlbx@mail.cis.hut.fi
http://www.cis.hut.fi/projects/somtoolbox/

April 18, 2000

Abstract

Self-Organizing Map is an unsupervised neural network which combines vector quantization and vector projection. This makes it a powerful visualization tool. SOM Toolbox implements the SOM in the Matlab 5 computing environment. In this paper, computational complexity of SOM and the applicability of the Toolbox are investigated. It is seen that the Toolbox is easily applicable to small data sets (under 10000 records) but can also be applied in case of medium sized data sets. The prime limiting factor is map size: the Toolbox is mainly suitable for training maps with 1000 map units or less.

Keywords

self-organizing map, batch map, computational complexity, data mining

Note: This is a corrected version of the paper. In the published version, the n_{V_j} term was missing from the denominator of Equation 5, the corresponding term `vn` was missing from the C-code of the batch training algorithm, and in both C-codes the operator `^` (which is actually bitwise XOR) has been changed to two commands which square the argument.

1 Introduction

The SOM Toolbox, hereafter simply called the Toolbox, is a function library for Matlab 5 computing environment implementing the Self-Organizing Map (SOM) algorithm, a popular neural network method based on unsupervised learning [2]. The Toolbox contains functions for the creation, visualization and analysis of Self-Organizing Maps.

In this paper, the SOM Toolbox (version 2) [6] is shortly presented and its scalability to data size is considered.

2 SOM Toolbox

The Toolbox is available free of charge under the GNU General Public License from the web site listed in the title part. However, to use it, one has to have Matlab (version 5.2 at least), by MathWorks, Inc. Matlab is a widely used programming environment for technical computing. It features a high-level programming language, powerful visualization, graphical user interface tools and a very efficient implementation of matrix calculus. These are major advantages in the data mining research because they allow fast prototyping, testing and customizing of the algorithms.

The Toolbox can be used to preprocess data, initialize and train SOMs using a range of different kinds of topologies, visualize SOMs in various ways, and analyze the properties of the SOMs and data, e.g. SOM quality, clusters on the map and correlations between variables.

The kind of data that can be handled with the Toolbox is so-called spreadsheet or table data. Each row of the table is one data sample. The items on the row are the variables, or components, of the data set. The table format is a very common data representation. The variables might be the properties of an object, or a set of measurements measured at a specific time.

An important benefit is that the data may contain missing values indicated with special `NaN` values in the Matlab matrix. In the SOM training algorithms missing components are handled by simply excluding them from the distance calculations. This is a valid approach since the same data sample is compared with all prototype vectors of the SOM, and thus the same components are always ignored.

The Toolbox can handle both numerical and symbolic data (ie. strings), but only the former is utilized in the SOM algorithm. Symbolic data can be inserted into string labels associated with each data sample. They can considered as post-it notes attached to each sample. One can check on them later to see what was the meaning of some specific sample, but the algorithm ignores them. If the symbolic variables need to be utilized in training the SOM, one can try converting them into numerical variables using, e.g., mapping or 1-of-n coding [5].

A deficiency of SOM Toolbox, and Matlab in general, is that the data must fit in the main memory. Also, Matlab has not been built to be conservative with respect to memory. For example, function parameters are always copied to the memory space of the function, thus creating an additional copy of the parameter. With very large matrices this wastes considerable amounts of memory.

3 SOM algorithms

The SOM is essentially a combined vector quantization and vector projection algorithm. It consists of neurons organized on a regular low-dimensional grid. Each neuron is represented by a d-dimensional weight vector $\mathbf{m}_i = [m_{i1}, \ldots, m_{id}]$, where d is equal to the dimension of the input vectors. The neurons are connected to adjacent neurons by a neighborhood relation, which dictates the topology, or structure, of the map. Typically the neurons are positioned on a 2-dimensional plane in a regular rectangular or hexagonal lattice.

3.1 Sequential training algorithm

The SOM is trained iteratively. In each training step, one sample vector $\mathbf{x}$ from the input data set is chosen randomly and the distance between it and all the weight vectors of the SOM is calculated using some distance measure. The neuron whose weight vector is closest to the input vector $\mathbf{x}$ is called the Best-Matching Unit (BMU):

$$||\mathbf{x} - \mathbf{m}_c|| = \min_i \{||\mathbf{x} - \mathbf{m}_i||\}, \tag{1}$$

where $||\cdot||$ is the distance measure, typically Euclidian distance. After finding the BMU, the weight vectors of the SOM are updated so that the BMU is moved closer to the input vector in the input space. The topological neighbors of the BMU are treated similarly. The update rule for the weight vector of the i is:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t) h_{ci}(t)[\mathbf{x}(t) - \mathbf{m}_i(t)], \tag{2}$$

$\mathbf{x}(t)$ is an input vector randomly drawn from the input data set , function $\alpha(t)$ is learning rate and t denotes time. The function $h_{ci}(t)$ is neighborhood kernel around the winner unit c.

3.2 Batch training algorithm

Batch Map is a variant of SOM. Instead of using a single data vector at a time, the whole data set is presented to the map before any adjustments are made — hence the name "batch". In each training step, the data set is partitioned according to the Voronoi regions of the map weight vectors, ie. each data vector belongs to the data set of the map unit to which it is closest. After this, the new weight vectors are calculated as:

$$\mathbf{m}_i(t+1) = \frac{\sum_{j=1}^{n} h_{ci}(t)\mathbf{x}_j}{\sum_{j=1}^{n} h_{ci}(t)}, \tag{3}$$

where n is the number of data samples and $c = arg\min_k\{||\mathbf{x}_j - \mathbf{m}_k||\}$ is the index of the BMU of data sample $\mathbf{x}_j$. The new weight vector is a weighted average of the data samples, where the weight of each data sample is the neighborhood function value $h_{ci}(t)$ at its BMU c. Alternatively, one can first calculate the sum of the vectors in each Voronoi set:

$$\mathbf{s}_i(t) = \sum_{j=1}^{n_{V_i}} \mathbf{x}_j, \tag{4}$$

where n_{V_i} is the number of samples in the Voronoi set. Then, the new values of the weight vectors can be calculated simply as:

$$\mathbf{m}_i(t+1) = \frac{\sum_{j=1}^{m} h_{ij}(t)\mathbf{s}_j(t)}{\sum_{j=1}^{m} n_{V_j} h_{ij}(t)}, \tag{5}$$

where m is the number of map units. This is the way batch algorithm has been implemented in the Toolbox, because it uses much less memory.

3.3 Training parameters

There are a number of training parameters that need to be decided before the training: map size (ie. the number of map units) and shape, neighborhood kernel function, neighborhood radius, learning rate and the length of training. The user can freely specify all of these, but to minimize user effort, the Toolbox also provides default values for them. The default values are:

- The number of map units is (approximately) $m = 5\sqrt{n}$, where n is the number of data samples.
- Map shape is rectangular sheet with hexagonal lattice. The ratio of sidelengths corresponds to the ratio between two greatest eigenvalues of the covariance matrix of the data.

- Neighborhood function is gaussian $h_{ci}(t) = e^{-\frac{\delta_{ci}^2}{2r(t)^2}}$, where δ_{ci} is the distance between units c and i on the map grid and $r(t)$ is the neighborhood radius at time t.

- Radius, as well as learning rate, is a monotonically decreasing function of time. The starting radius depends on map size, but the final radius is one. Learning rate starts from 0.5 and ends to (almost) zero.

- Training length is measured in epochs: one epochs corresponds to one pass through the data. The number of epochs is directly proportional to the ratio between number of map units and number of data samples m/n.

By default, the training is divided to two phases: rough training and fine-tuning. The first phase is performed using bigger neighborhood radius and learning rate than the second phase. It is also shorter than the second phase.

3.4 Computational complexity

As C-code, the sequential training algorithm can be implemented as:

```
for (j=0; j<n; j++) {           /* go through the data */
  bmu=-1; min=1000000;
  for (i=0; i<m; i++) {         /* find the BMU */
    dist=0;
    for (k=0; k<d; k++) {diff = X[j][k] - M[i][k]; dist += diff*diff; }
    if (dist<min) { min=dist; bmu=i; }
  }
  for (i=0; i<m; i++) {         /* update */
    h = alpha*exp(delta(bmu,i)/r);
    for (k=0; k<d; k++) M[i][k] -= h*(M[i][k] - X[j][k]);
  }
}
```

where `X[j][k]` is the kth component of jth data sample, `M[i][k]` is the kth component of map unit i and `delta` is a table of squared map grid distances between map units calculated beforehand. The radius `r` corresponds to $-2r(t)^2$ above. Correspondingly, the batch training algorithm is:

```
for (i=0; i<m; i++) { vn[i] = 0; for (k=0; k<d; k++) S[i][k] = 0; }
for (j=0; j<n; j++) {           /* go through the data */
  bmu=-1; min=1000000;
  for (i=0; i<m; i++) {         /* find the BMU */
```

```
    dist=0;
    for (k=0; k<d; k++) { diff = X[j][k] - M[i][k]; dist += diff*diff; }
    if (dist<min) { min=dist; bmu=i; vn[bmu]++; }
  }
  for (k=0; k<d; k++) S[bmu][k] += X[j][k]; /* Voronoi region sum */
}
for (i=0; i<m; i++) for (k=0; k<d; k++) M[i][k] = 0;
for (i1=0; i1<m; i1++) {        /* update */
  htot = 0;
  for (i2=0; i2<m; i2++) {
    h = exp(delta[i1][i2]/r);
    for (k=0; k<d; k++) M[i1][k] += h*S[i2][k];
    htot += h*vn[i2];
  }
  for (k=0; k<d; k++) M[i1][k] /= htot;
}
```

Thus, there are $6nmd + 2nm$ floating point operations (additions, substractions, multiplications, divisions or exponents) in the sequential algorithm and $3nmd + (2d + 5)m^2 + (n + m)d$ operations in the batch training algorithm. If $n \gg m$ this means that the computational complexity of one epoch of sequential training is about $\mathcal{O}(nmd)$ and that the computational complexity of batch training is about half of the sequential one.

The memory consumption of sequential algorithm is $(n+m)d$ floating points for data and map prototype vector matrices. For batch algorithm, memory consumption is $(n+2m)d$ floating points for data, prototype and sum vector matrices. In addition, in both cases there is the interunit distances calculated beforehand, which is a matrix of m^2 elements, although this can be reduced to $m(m-1)/2$ floating points since the matrix is symmetric. Here it is assumed that the data is in the main memory. This need not be so. For example `SOM_PAK` [3] can use buffered data.

In the Toolbox, things are a bit different from the above C-code. First of all, the distance metric is slightly different from Euclidian: the distance calculation must take possible missing values into account, and the Euclidian distance is weighted:

$$||\mathbf{x} - \mathbf{m}||_\Lambda^2 = (\mathbf{x} - \mathbf{m})^T \Lambda (\mathbf{x} - \mathbf{m}), \qquad (6)$$

where Λ is a diagonal matrix. This way the different variables can be weighted according to their importance. Both operations increase the computational complexity somewhat, but the algorithm still remains essentially $\mathcal{O}(nmd)$ in complexity.

If default parameter values listed in the previous section are used, one can also calculate the complexity of the whole training process in the Toolbox. The number of map units is about $m = 5\sqrt{n}$ and the number of epochs is proportional to m/n. Thus the total complexity is $n(5\sqrt{n})d(5\sqrt{n})/n$ or $\mathcal{O}(nd)$. Of course, in some cases the number of map units needs to be selected differently, e.g. $m = 0.1n$ in which case the complexity is $\mathcal{O}(n^2d)$.

There are also some faster variants of the SOM, for example the TS-SOM [4]. These are basically based on speeding up the winner search from $\mathcal{O}(md)$ to $\mathcal{O}(log(m)d)$ by investigating only a small number of prototypes.

4 Experiments

While investigation of the number of operations in the algorithms gives useful insight to the compulational load, it does not give information of the actual computation times of the Toolbox. In order to get an idea of how efficient the Toolbox implementation is, and how the map size and length of data effect the computing time, some performance tests were made. The purpose was only to evaluate the computational load of the algorithms. No attempt was made to compare the quality of the resulting mappings, primarily because there is no uniformly recognized "correct" method to evaluate it.

4.1 Test set-up

The tests were run in a machine with 3 GBs of memory and 16 250 MHz R10000 64-bit CPUs (one of which was used by the test process) running IRIX 6.5 operating system. The Matlab version was 5.3. The different parameters of the test (data size, training length, etc.) are listed in Table 1. The test parameters included different data set and map sizes and three training functions: `som_batchtrain`, `som_seqtrain` and `vsom`, the last of which is part of the `SOM_PAK` package [3]. The training length, 10 epochs, is fairly standard, although with very large data sets it is perhaps excessive.

The computing times measured for `vsom` do not include the time needed for writing and reading the files from Matlab. This took between 0.2 and 50 seconds, depending primarily on the size of the data set. Especially with large maps, this is irrelevantly small time when compared with overall training time. However, the measured times do contain the time used by `vsom` itself for file I/O. Thus, there is some small overhead in the computing times for `vsom` with respect to computing times of `som_batchtrain` and `som_seqtrain`.

Table 1: Performance test parameters.

parameter	different values used in the test
data dimension	10, 30, 50, 100
data length	300, 1000, 3000, 10000, 30000
number of map units	30, 100, 300, 1000
training function	`som_batchtrain`
	`som_seqtrain`
	`vsom`
neighborhood function	gaussian
training length	10 epochs

4.2 Results

Figure 1 shows test results. For a "typical" data of size `[3000 x 30]` and 300 map units, the training times were 8, 77 and 43 seconds, for `som_batchtrain`, `som_seqtrain` and `vsom`, respectively. For the largest investigated case of `[30000 x 100]` data and 1000 map units the times were 8 minutes, 2.2 hours and 47 minutes, respectively.

It can be seen that the batch training algorithm is almost always considerably faster than the others: upto 20 times faster than `som_seqtrain` and upto 11 times faster than `vsom`, with mean values being 11 and 4. Likewise, sequential training is almost always the slowest algorithm. The only case where this is not so is when the number of map units exceeds the number of data samples $n < m$ (top right corner of Figure 1) — a very unlikely case in practice. The superiority of `som_batchtrain` over `som_seqtrain` is no suprise, but it was suprising to find that the C-program `vsom` was considerably slower than `som_batchtrain`.

Figure 2 shows some typical computing times as a function of the number of data samples, map units and data dimension. Computing time scales almost linearly with respect to dimension and number of data samples (on this parameter range), but grows faster with the number of map units. This corresponds well to the results in Section 3.4.

4.3 Additional tests

The gaussian neighborhood function is computationally quite heavy as opposed to the bubble neighborhood function. However, with respect to the rest of the algorithm, this seems to have rather small effect in Matlab. However,

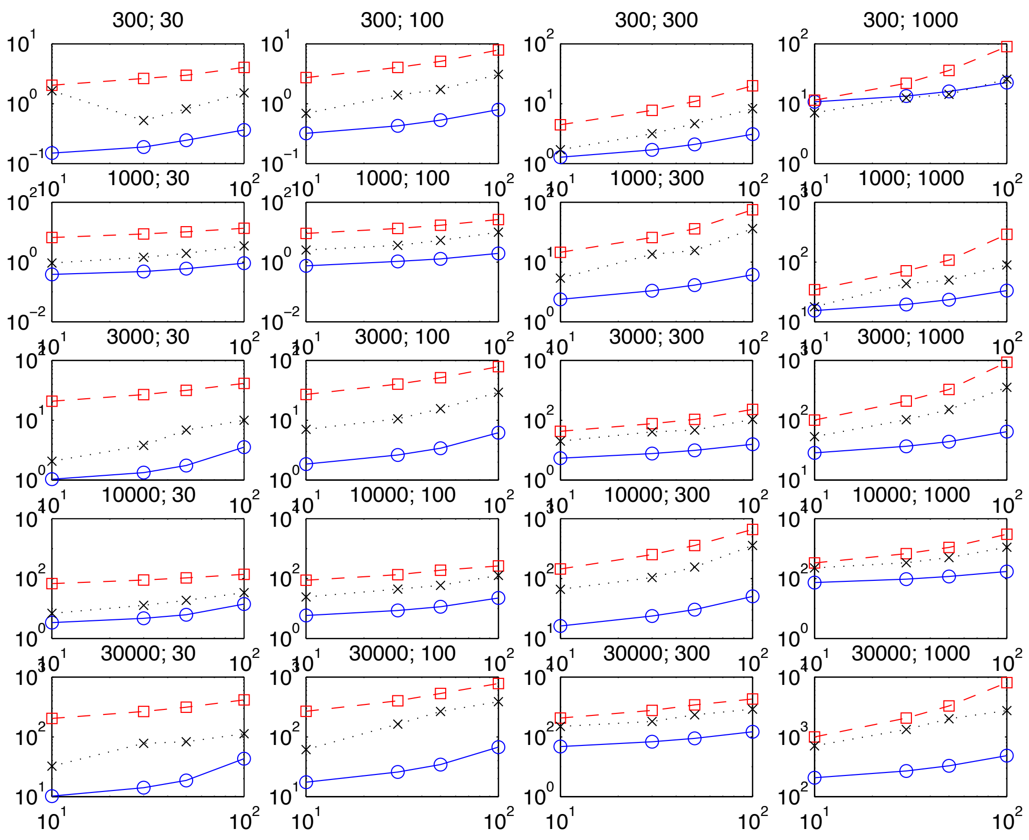

Figure 1: Computing time (in seconds) as a function of data dimension. Both axis are logarithmic. The subplot titles $n; m$ show the number of data samples (n) and map units (m). Times for batch training algorithm are shown with circles, times for sequential training with squares and times for `vsom` with crosses.

in `vsom` the effect was more significant, dropping the training time by about 30% on the average.

One of the weak points of Matlab is that loops are relatively slow when compared with precompiled code (like C-programs). Matlab has a compiler that can be used to precompile Matlab functions into so-called mex-files (short for Matlab executable), but this is really beneficial only if there are a lot of loops in the function. For this reason it is natural that we observed some benefits with respect to `som_seqtrain` (in the order of 20%; the longer the training, the greater the benefit) but none with respect to `som_batchtrain`.

Some tests were also performed in a workstation with a single 350 MHz Pentium II 32-bit CPU and 256 MBs of memory. The tests were performed both in Linux and Windows NT operating systems. The relative computation

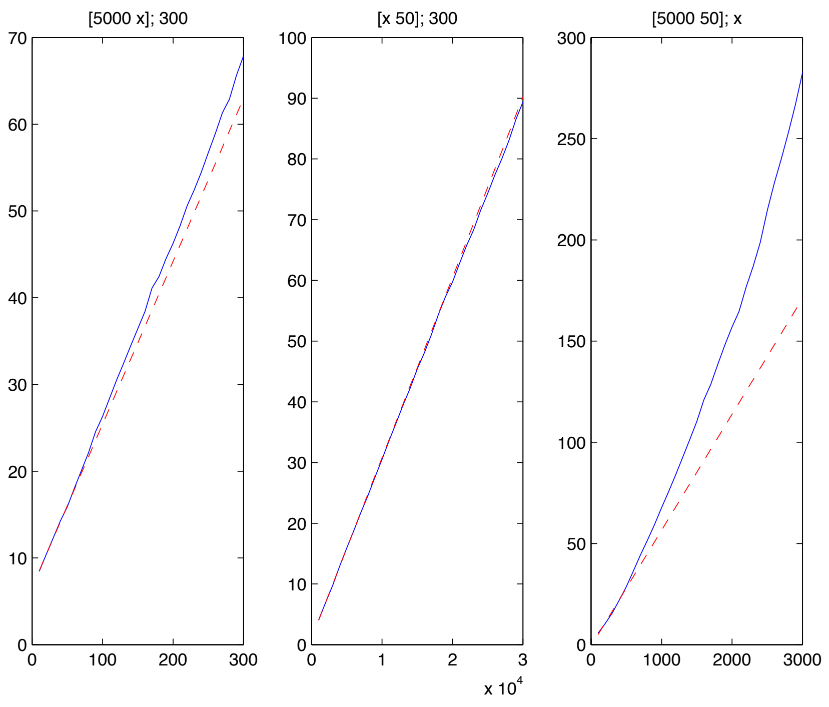

Figure 2: Computing time of `som_batchtrain` (in seconds) as a function of the number of data samples (on the left, with 300 map units, dimension 50), number of map units (in the center, with 5000 data samples, dimension 50) and data dimension (on the right, with 5000 data samples and 300 map units). The dashed line is a 1st order polynomial fitted to the using five first samples.

times for IRIX, Linux and Windows respectively were

- 1:5:2.5 for `som_batchtrain`
- 1:3.3:2.7 for `som_seqtrain`
- 1:1.7:4 for `vsom`

Of course, these results reflect differences between underlying hardware, operating system and optimization of the Matlab itself in different platforms and it is impossible to say how much each factor affected the results. In the workstation, the lower amount of memory certainly had an effect with large data and map sizes. Since the performace of `vsom` in Linux was not effected by this, with large maps `vsom` was approximately as fast or even faster than `som_batchtrain`. The slowness of `vsom` in Windows may own something to the fact that the Windows version of `vsom` was compiled back in 1996, while the Unix versions were compiled just recently.

4.4 Memory

The major deficiency of the SOM Toolbox is the expenditure of memory. A rough estimate of the amount of memory used by `som_batchtrain` is given by: $8 \times (5md + 4nd + 3m^2)$ bytes, where m is the number of map units, n is the number of data samples and d is the input space dimension. Especially the last term limits the usability of the Toolbox considerably. Consider a relatively small data set `[3000 x 10]` and 300 map units. The amount of memory required is still moderate, in the order of 3 MBs. However, increasing the map size to 3000 map units, the memory requirement is almost 220 MBs, 99% of which comes from the last term of the equation. The sequential algorithm is less extreme requiring only one half or one third of this.

5 Applicability of the Toolbox

In [1] Goebel and Gruenwald prodive a survey of data currently available data mining tools. Below is a characterization of the Toolbox (together with Matlab) as a data mining tool following their categorization scheme.

5.1 General characteristics

SOM Toolbox is freeware, beta-status product directly downloadable from the Internet. However, to run it requires Matlab. The general characteristics of Matlab are: a commercial product, with academic licensing fees. One can download an evaluation version of Matlab from the Internet. The computer architecture is either standalone or client/server. Matlab is available for Windows and various Unix platforms.

5.2 Database connectivity

The basic Matlab is not particularly strong in database connectivity. About the only data format supported is ASCII. However, there are some toolboxes which ameliorate this lack somewhat. For example, Database Toolbox enables connection to ODBC database using SQL commands, and Excel Link allows data exchange with Excel. The analysis is typically performed offline, but also online connection with database is possible.

The data model in SOM Toolbox, and also in Matlab, a single table. The maximum number of records that SOM Toolbox can comfortably handle depends on the available hardware. Goebel and Gruenwald divide the data size to three categories: small (upto 10000 records), medium (10000 - 1000000 records) and large (over million records). Considering the tests in the previous section, SOM Toolbox is comfortable to use with small data sets, but can also handle medium data sets.

Matlab can accommodate continuous, categorial and symbolic data. Also SOM Toolbox can handle all of these, but the algorithm really uses only numerical data, so with symbolic variables the Toolbox is of little use.

5.3 Data mining characteristics

SOM Toolbox is primarily a clustering and visualization tool, and thus is an excellent tool for exploratory data analysis. However, it can be used in most other discovery tasks as well: preprocessing, prediction, regression and classification. The discovery methodology is neural networks. Matlab itself is a powerful preprocessing and visualization environment. There are a large number of toolboxes intended for a variety of modeling and analysis tasks. These toolboxes are based on wide span of methodologies from statistical methods to bayesian networks.

In Matlab, the data mining process is typically highly interactive. However, for specific tasks and applications it is possible to build partly or totally autonomous tools. SOM Toolbox is also most useful with high degree interaction, although default values can be used for most parameters. However, simply because SOM Toolbox is primarily intended as a tool for data understanding, it can never become a completely autonomous tool.

6 Conclusion

In this paper, the scalability and applicability of the SOM Toolbox in data mining has been considered. The Toolbox runs in Matlab, so while the Toolbox itself is free, it is not zero-budget software. Matlab as a computing environment is extremely versatile but for this reason also requires high degree of user interaction.

The data model in Toolbox is a single table with continuous or categorial variables. The Toolbox is easily applicable to small data sets (under 10000 records) but can also be applied in case of medium sized data sets (upto 1000000 records). Here, a moderate number of variables — under a few hundred at most — is assumed. An important limiting factor is however map size. The Toolbox is mainly suitable for training maps with 1000 map units or less. If this kind of parameter values are used, especially the batch training algorithm is very efficient, being considerably faster than `vsom`, a commonly used implementation of sequential SOM algorithm in C-code.

In data mining, the Toolbox and the SOM in general can be used for a wide range of tasks from preprocessing to modeling. However, it is best suited for the data understanding: clustering, visualization and exploratory data analysis.

Acknowledgments

This work has been mostly carried out in “Adaptive and Intelligent Systems Applications” technology program of Technology Development Center of Finland, and the EU financed Brite/Euram project “Application of Neural Network Based Models for Optimization of the Rolling Process” (NEUROLL). The cooperation and financing of Jaakko Pöyry Consulting, Rautaruukki Raahe Steel, Laboratory of Metallurgy at Helsinki University of Technology and the Technology Development Center of Finland (TEKES) are gratefully acknowledged.

References

[1] Goebel, M. and Gruenwald L., A survey of data mining and knowledge discovery software tools, *SIGKDD Explorations*, Vol. 1, June (1999).

[2] Kohonen, T., *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, Berlin, Heidelberg (1995).

[3] Kohonen, T., Hynninen, J., Kangas, J. and Laaksonen, J., SOM_PAK: The Self-Organizing Map Program Package, Technical Report A31, Helsinki University of Technology (1996).

[4] Koikkainen, P., Fast Deterministic self-organizing maps, In *Proceedings of ICANN'95, International Conference on Artificial Neural Networks*, Vol II, pages 63–68, 1995.

[5] Pyle, D., *Data Preparation for Data Mining*. Morgan Kaufmann Publishers (1999).

[6] Vesanto, J., Himberg, J., Alhoniemi, E. and Parhankangas, J., Self-organizing map in Matlab: the SOM Toolbox. In *Proceedings of the Matlab DSP Conference 1999*, pages 35–40, Espoo, Finland (1999).