Self Organizing Maps with Delay Actualization

Lukáš Vojáček[1,2(✉)], Pavla Dráždilová[2], and Jiří Dvorský[2]

[1] IT4Innovations, VŠB - Technical University of Ostrava, 17. listopadu 15/2172, 708 33 Ostrava, Czech Republic
lukas.vojacek@vsb.cz

[2] Department of Computer Science, VŠB – Technical University of Ostrava, 17. listopadu 15/2172, 708 33 Ostrava, Czech Republic
{pavla.drazdilova,jiri.dvorsky}@vsb.cz

Abstract. The paper deals with the Self Organizing Maps (SOM). The SOM is a standard tool for clustering and visualization of high-dimensional data. The learning phase of SOM is time-consuming especially for large datasets. There are two main bottleneck in the learning phase of SOM: finding of a winner of competitive learning process and updating of neurons' weights. The paper is focused on the second problem. There are two extremal update strategies. Using the first strategy, all necessary updates are done immediately after processing one input vector. The other extremal choice is used in Batch SOM – updates are processed at the end of whole epoch. In this paper we study update strategies between these two extremal strategies. Learning of the SOM with delay updates are proposed in the paper. Proposed strategies are also experimentally evaluated.

Keywords: Self organizing maps · High-dimensional dataset · High performance computing

1 Introduction

Recently, the issue of high-dimensional data clustering has arisen together with the development of information and communication technologies which support growing opportunities to process large data collections. High-dimensional data collections are commonly available in areas like medicine, biology, information retrieval, web analysis, social network analysis, image processing, financial transaction analysis and many others.

Two main challenges should be solved to process high-dimensional data collections. One of the problems is the fast growth of computational complexity with respect to growing data dimensionality. The second one is specific similarity measurement in a high-dimensional space. Beyer et al. presented in [1] that for the expected distance any point in a high-dimensional space, computed by the Euclidean distance to the closest and to the farthest point, shrinks with growing dimensionality. These two reasons reduce the effectiveness of clustering algorithms on the above-mentioned high-dimensional data collections in many actual applications.

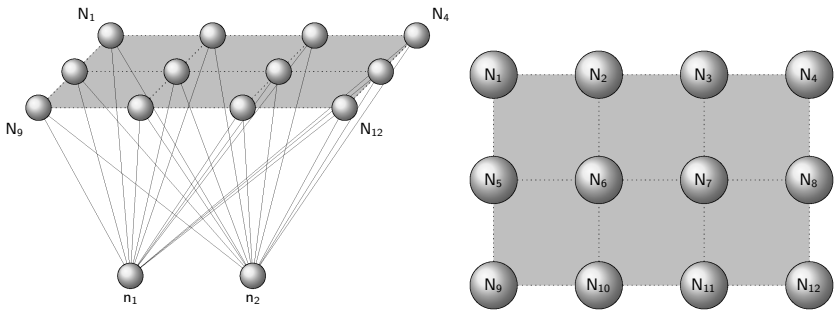
K. Saeed and W. Homenda (Eds.): CISIM 2015, LNCS 9339, pp. 154–165, 2015.
DOI: 10.1007/978-3-319-24369-6_13

The paper is organized as follows. In Sect. 2 we will describe one Self Organizing Maps. Section 3 describes parallel design of SOM learning algorithm. Modification of weights' update process is given in Sect. 4. Some experimental results are presented in Sect. 5. The paper is summarized and conslusions are made in Sect. 6.

2 Self Organizing Maps

Self Organizing Maps (SOMs), also known as Kohonen maps, were proposed by Teuvo Kohonen in 1982 [3]. SOM consists of two layers of neurons: an *input layer* that receives and transmits the input information, and an *output layer*, that represents the output characteristics. The output layer is commonly organized as a two-dimensional rectangular grid of nodes, where each node corresponds to one neuron. Both layers are feed-forward connected. Each neuron in the input layer is connected to each neuron in the output layer. A real number, *weight*, is assigned to each of these connections. i.e. weights of all connections for given neuron form *weight vector*. SOM is a kind of artificial neural network that is trained by unsupervised learning. Learning of the SOM is competitive process, in which neurons compete for the right to respond to a training sample. The winner of the competition is called *Best Matching Unit* (BMU).

Using SOM, the input space of training samples can be represented in a lower-dimensional (often two-dimensional) space [4], called a *map*. Such a model is efficient in structure visualization due to its feature of topological preservation using a neighbourhood function.



(a) Global view on SOM structure (b) SOM output layer

Fig. 1. Basic Schema of SOM

3 Parallel SOM Learning Algorithm

A *network partitioning* is the most suitable implementation of the parallelization of an SOM learning algorithm. Network partitioning is an implementation of the

learning algorithm, where the neural network is partitioned among the processes. Network partitioning has been implemented by several authors [2,9]. The parallel implementation proposed in this work is derived from the standard sequential SOM learning algorithm.

After analysing the serial SOM learning algorithm we have identified the two most processor time-consuming areas. These parts were selected as candidates for the possible parallelization. The selected areas were:

Finding BMU – this part of SOM learning can be significantly accelerated by dividing the SOM output layer into smaller pieces. Each piece is then assigned to an individual computation process. The calculation of Euclidean distance among the individual input vector and all the weight vectors to find BMU in a given part of the SOM output layer is the crucial point of this part of SOM learning. Each process finds its own, partial, BMU in its part of the SOM output layer. Each partial BMU is then compared with other BMUs obtained by other processes. Information about the BMU of the whole network is then transmitted to all the processes to perform the updates of the BMU neighbourhood.

Weight Actualization – Weight vectors of neurons in the BMU neighbourhood are updated in this phase. The updating process can also be performed using parallel processing. Each process can effectively detect whether or not some of its neurons belong to BMU neighbourhood. If so, the selected neurons are updated.

A detailed description of our approach to the parallelization process is described in Fig. 2.

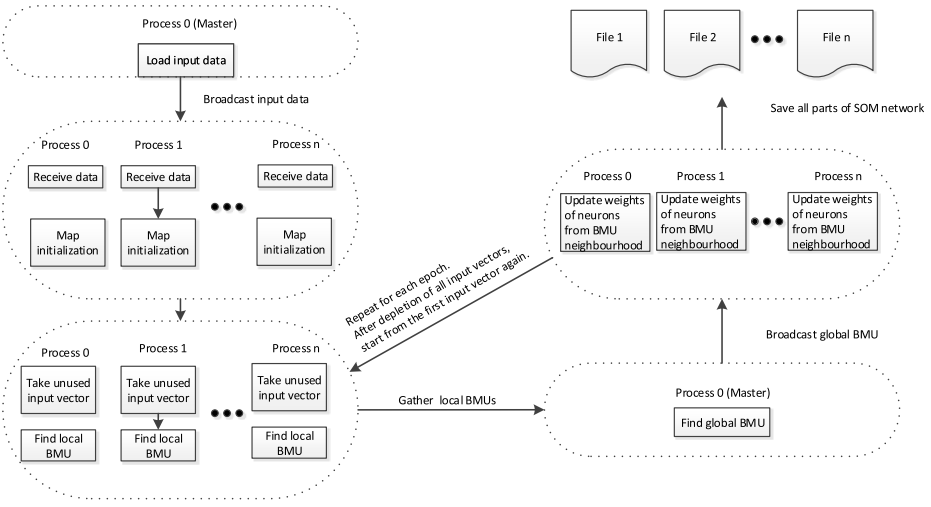


Fig. 2. Improved Parallel SOM Algorithm

Before any implementation of an experimental application began, we had to decide how the parallelization would be done. Initially, we supposed that the most effective approach is to divide the SOM into several parts or blocks, where each block is assigned to the individual computational process. For example, let's suppose that an SOM with neurons $N = 20$ in the output layer is given. The output layer is formed as a rectangular grid with number of rows $N_r = 4$ and number of columns $N_c = 5$. Then the output layer of the SOM is divided into 3 continuous blocks which are associated with three processes[1].

To remove the unbalanced load, the approach to the parallelization process has been modified. The division of the SOM output layer was changed from a block load to a cyclic one. The individual neurons were assigned to the processes in a cyclic manner. A nearly uniform distribution of the output layer's neurons among processes is the main advantage of this kind of parallelization. The uniform distribution of the neurons plays an important role in weight actualization because there is a strong assumption that neurons in the BMU neighbourhood will belong to different processes. An example of a cyclic division of the SOM output layer with a dimension of 4×5 neurons can be seen in Fig. 3, where each neuron is labeled with a color of assigned process. A more detailed description of parallelization can be found in our previous papers (including a full notation)[6,8].

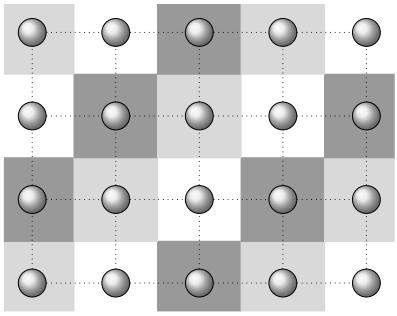


Fig. 3. Cyclic Division

4 Delay Actualization

In this modification of the SOM algorithm we focused on the area called *finding BMU*. Only in the parallel version is it necessary to find the global BMU from the local BMUs in each iteration and here are two areas, by which we will discuss:

1. To find the global BMU we must transfer a lot of data between processes.

[1] If there is no possibility of dividing the output layer into a block with the same number of neurons, some blocks have one extra neuron.

2. This waiting mode (blocking communication), where other processes and threads awaiting the outcome, will decrease the efficiency of parallel computation.

The method described below is based on information that with the same amount of data it is effective to send data all at once instead of sending in portions. Both problems mentioned above are solved this way. The proof about transfer data is in Table 1, where 1 to 64 processes are used and transferred 50 thousand and 500 thousand numbers of all processes on a single process, but in one case we send all of these numbers together and in the second case separately - at one time two numbers together. For clarity, the number of data that are transfer from individual processes is always the same. Only the total number of data that are finally placed on the target process is changing. For example, if we have 6 processes and 50k numbers, so we have on the target process saved 6×50k numbers. From these results it is possible to see that the final times for both amounts (50k and 500k) and for sending data together are the very similar.

For data transmission, the MPI functions *Gather*[2] are used and the processes are running on separate computing nodes, which are connected by the Infiniband network.

Table 1. Comparison of data transmission

Processes	Time [sec]			
	50k		500k	
	Separate data	Together data	Separate data	Together data
1	0.0199089	0.0000188	0.197716	0.000154018
2	0.146948	0.030802	2.06079	0.031806
4	0.316004	0.062149	3.14302	0.0727129
8	0.487644	0.142774	5.09625	0.153242
16	0.67049	0.304665	6.58546	0.30529
32	0.785121	0.627432	8.63451	0.633787
64	0.959224	1.26573	9.81402	1.25528

The second point, which we mentioned above, concerns the utilization of individual processes or threads (both parallelization operate on the same principle, see previous article [6]). As we mentioned earlier we divide the SOM algorithm into two parts: The first part concerns the search for the BMU (the fast part) and the second part concerns updating weight (time-consuming part). The delay occurs in the situation where some processes (threads) must update more neurons than the other processes (threads), an example can be seen in Fig. 4. Where the process number two must update three neurons, but other processes must update only two neurons. If the update does not occur after each iteration, but only after a certain time, it is possible to reduce the impact of blocking communication. Individual processes will not have to wait for other processes and utilization processes should be uniform. It is because of two reasons:

1. A BMU is usually different with each iteration and therefore neurons which must be updated are different.
2. The number of neurons to be updated decreases, but at the beginning of the algorithm 1/4 of all neurons are updated. An important factor that affects the distribution is training data and unfortunately this cannot be anticipated.

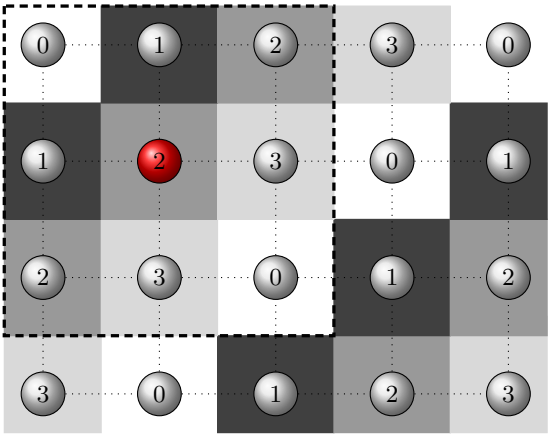


Fig. 4. Example of actualization area

According to the above examples our goal is aggregate data which are transmitted and we propose the following approach to update the weights:

The base is the parallel solution which we described in Sect. 3, where at the beginning limit of delay – L is set for how many local BMU can be kept in the local memory in each process. Each processes find the local BMU and save the result in the local memory. If the limit is not reached, it is necessary to read the new input vector and find a new local BMU. If the limit is reached all local BMUs are moved to a process with rank 0, which finds a global BMU for each iteration and then sends results to all processes. After this step each process gradually updates the weights.

We worked with three variants of the above described algorithm:

1. *Constant delay* (*Cons*) – Size of L is the same throughout the calculation.

2. *Decreasing delay* (*Dec*) – Size of L is decreasing by ζ at the start of each epoch to 1.

3. *Increasing delay* (*Inc*) – The value of how many local BMUs can be kept in the local memory is set at 1 and increases by ζ at the end of the each epoch until it reaches L.

For a complete description of the algorithm is to be noted that ζ is applied at the end of each epoch (only for variants *Dec* and *Inc*). Setting the value of ζ, for variants *Dec* and *Inc* largely depends on the number of input vectors M. Therefore we are working with a percentage of M. It is used for settings L and ζ. In the chapter experiments we attempted to show how much influence the value of ζ is. For example: $L = 10\%$ of M, $\zeta = 0.1\%$ of M.

Here it is necessary to briefly recall the behaviour of the neural network SOM. Over time, the number of updated neurons is changing – decreases. At the beginning, most of the neurons are updated but at the end only the few neurons or only one neuron are. If variant *Dec* is used, delay gradually decreases by ζ and also the number of neurons that must be updated. In variant *Inc* it is the opposite, by the number of updated neurons still decreases but the delay increases.

5 Experiments

We will describe different datasets and we will provide experiments with bigger and smaller limit of delay – L. The mean quantization error (MQE) is used to compare the quality of the neural network method which it is described in paper [5].

5.1 Experimental Datasets and Hardware

Weblogs Dataset. A Weblogs dataset was used to test learning algorithm effectiveness on high dimensional datasets. The Weblogs dataset contained web logs from an Apache server. The dataset contained records of two months worth of requested activities (HTTP requests) from the NASA Kennedy Space Center WWW server in Florida[2]. Standard data preprocessing methods were applied to the obtained dataset. The records from search engines and spiders were removed, and only the web site browsing option was left (without download of pictures and icons, stylesheets, scripts etc.). The final dataset (input vector space) had a dimension of 90,060 and consisted of 54,961 input vectors. For a detailed description, see our previous work [7], where web site community behaviour has been analyzed.

On the base of this dataset 15,560 user profiles were extracted and the number of profile attributes is 28,894 (this number corresponds to the dimension of input space) for the final dataset.

Experimental Hardware. The experiments were performed on a Linux HPC cluster, named Anselm, with 209 computing nodes, where each node had 16 processors with 64 GB of memory.The processors in the nodes were Intel Sandy

[2] This collection can be downloaded from http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

Bridge E5-2665. Compute network is InfiniBand QDR, fully non-blocking, fat-tree. Detailed information about hardware can be found on the website of Anselm HPC cluster[3].

In this section we describe experiments which are based on delay actualizations. For the experiment, we examine the quality of the resulting neural networks and the time that is required for calculation. The type of parallelization of SOM is a combination of MPI and OpenMP.

5.2 First Part of the Experiment

The first part of the experiments was oriented towards an examinination of the quality of neural networks which depends on the size of the delay. The dataset used is Weblogs. All the experiments in this section were carried out for 1000 epochs; the random initial values of neuron weights in the first epoch were always set to the same values. The tests were performed for SOM with rectangular shape – 400×400 neurons. All three variants shown in section 4, are tested. If variants *Inc* or *Dec* are used, then the steps ζ are as follows 0.1%, 0.01%, 0.005%. MQE errors are presented for limit of delays L equal 5%, 10% and 20% in Table 2. Step size does not affect the variant *Cons*. Therefore, this method has only one value instead of three in the above table.

Table 2. MQE depends to limit of delay actualization

Steps [%]	MQE								
	Delay L = 5%			Delay L = 10%			Delay L = 20%		
	Inc	Dec	Cons	Inc	Dec	Cons	Inc	Dec	Cons
0.100	1.71474	0.54016	1.71477	1.7792	0.55492	1.77917	1.79262	0.56304	1.87535
0.010	1.66801	0.55741	1.71477	1.68754	1.10173	1.77917	1.69227	1.68584	1.87535
0.005	1.66795	1.08419	1.71477	1.67759	1.67199	1.77917	1.68192	1.69745	1.87535

5.3 Second Part of the Experiment

The second part of the experiments were oriented towards scalability. As in the previous test, experiments are carried out on three types of delays (increasing, decreasing and constant). The parallel version of the learning algorithm was run using 16, 32, 64, 128, 256, 512, and 1024 cores respectively. The achieved computing time is presented in the Table 3 for step $\zeta = 0.1\%$, in Table 4 for step $\zeta = 0.01\%$, in Table 5 for step $\zeta = 0.005\%$. In the above tables the variant *Cons* is presented (it is not affected by the step – all three tables contains same values), the reason is comparison resulting times. For comparison, the standard SOM algorithm (without any delay) takes 32:10:30 computing time and MQE is 0.4825.

[3] https://support.it4i.cz/docs/anselm-cluster-documentation/hardware-overview

Table 3. Scalability of delay actualization – step $\zeta = 0.1\%$

Cores	Computing Time [hh:mm:ss]								
	Delay L = 5%			Delay L = 10%			Delay L = 20%		
	Inc	Dec	Cons	Inc	Dec	Cons	Inc	Dec	Cons
16	12:24:19	27:04:58	12:13:49	12:17:21	23:47:34	11:53:06	12:11:46	18:35:00	11:47:09
32	5:11:19	13:11:54	5:05:59	5:10:21	11:30:02	5:00:52	5:32:45	8:41:41	5:04:13
64	1:45:20	6:13:49	1:42:12	1:43:16	5:14:24	1:38:14	1:51:56	3:43:40	1:39:23
128	0:45:11	3:15:36	0:43:53	0:43:37	2:44:05	0:41:10	0:47:00	1:53:40	0:40:44
256	0:24:08	1:57:26	0:23:18	0:22:57	1:38:43	0:21:13	0:23:43	1:10:33	0:20:50
512	0:15:53	1:24:03	0:15:21	0:14:34	1:11:20	0:13:18	0:15:53	0:53:07	0:12:59
1024	0:17:34	1:25:12	0:16:54	0:15:36	1:17:37	0:14:23	0:17:25	1:03:08	0:14:05

Table 4. Scalability of delay actualization – step $\zeta = 0.01\%$

Cores	Computing Time [hh:mm:ss]								
	Delay L = 5%			Delay L = 10%			Delay L = 20%		
	Inc	Dec	Cons	Inc	Dec	Cons	Inc	Dec	Cons
16	13:54:21	14:19:45	12:13:49	13:18:46	13:04:25	11:53:06	13:20:08	12:21:52	11:47:09
32	5:28:27	6:17:04	5:05:59	5:36:31	5:29:52	5:00:52	6:03:48	5:36:57	5:04:13
64	1:54:03	2:20:04	1:42:12	1:57:10	1:57:31	1:38:14	2:12:01	1:57:21	1:39:23
128	0:49:31	1:10:31	0:43:53	0:50:53	0:53:14	0:41:10	0:59:19	0:50:42	0:40:44
256	0:24:29	0:43:56	0:23:18	0:24:48	0:29:53	0:21:13	0:29:52	0:24:51	0:20:50
512	0:15:42	0:35:07	0:15:21	0:15:57	0:21:29	0:13:18	0:19:33	0:16:00	0:12:59
1024	0:17:21	0:46:27	0:16:54	0:16:36	0:25:57	0:14:23	0:21:34	0:16:50	0:14:05

Table 5. Scalability of delay actualization – step $\zeta = 0.005\%$

Cores	Computing Time [hh:mm:ss]								
	Delay L = 5%			Delay L = 10%			Delay L = 20%		
	Inc	Dec	Cons	Inc	Dec	Cons	Inc	Dec	Cons
16	13:09:47	13:01:33	12:13:49	14:09:51	12:24:42	11:53:06	14:09:44	12:11:29	11:47:09
32	6:03:58	5:31:41	5:05:59	6:03:42	5:14:13	5:00:52	6:00:06	5:10:09	5:04:13
64	2:12:38	1:53:28	1:42:12	2:12:17	1:47:21	1:38:14	2:10:08	1:42:41	1:39:23
128	0:59:15	0:53:32	0:43:53	0:59:27	0:44:41	0:41:10	0:57:51	0:42:06	0:40:44
256	0:31:50	0:28:44	0:23:18	0:30:36	0:23:02	0:21:13	0:28:24	0:21:46	0:20:50
512	0:21:17	0:19:46	0:15:21	0:20:01	0:14:46	0:13:18	0:17:56	0:13:46	0:12:59
1024	0:22:38	0:22:14	0:16:54	0:21:46	0:16:07	0:14:23	00:19:04	0:14:06	0:14:05

Conclusion of Delay Experiments. In this section an evaluation of the above described experiments can be found. The reason this evaluation is discussed in a separate part is that the overall evaluation of effectiveness can not only be based on individual results. It is necessary to focus on a combination of outcomes for finding the optimal solution.

From the first experiment, which was focused on the quality of the final neural network, we can deduce the following conclusions:

1. As we expected, with the increasing size of the local memory, the overall quality of the neural networks is deteriorating. This behavior is evident in all three types of delays.

2. The variant *Cons* was in all three cases the worst.

3. According to these results, the variants *Inc* and *Dec* fundamentally differ from each other. When we use the variant *Dec*, the subsequent decrease of the value of the delay deteriorates the quality of the neural networks, but when we use the variant *Inc*, the quality of the neural network improves; it is not a significant change in the same way as in the *Dec*.

The second experiment was focused on the scalability and the time consumption of the above variants. We describe the results of the experiments as follows:

1. Even though the variant *Cons* is independent of the value steps ζ it still achieves the fastest computing time.

2. When the variant *Cons* and the variant *Inc* are used, the time difference between the delay (5%, 10% and 20%) is only a small percent - almost negligible. However, the variant *Dec* reaches time differences of up to 60%.

3. When 16 cores are used and step $\zeta = 0.1\%$ so the variant *Inc* is much faster (more than twice) than the variant *Dec*. Again, using 16 cores and at step $\zeta = 0.01\%$ times are in both the above variants almost comparable. However, when step $\zeta = 0.005\%$ is used, the variant *Dec* is slightly faster than the variant *Inc*.

If we look only at the individual results according to the first experiment, the overall best results are obtained with the variant *Dec* (delay $L = 5\%$ and step $\zeta = 0.1\%$) and the worst results are obtained with the variant *Cons*. The second experiment shows that the variant *Cons* is the fastest and the variation *Inc* is minimally affected by the delay amount. After comparing all the achieved results and the required time to calculate them, we have identified as the best variant *Dec* (delay $L = 5\%$ and step $\zeta = 0.01\%$) with computing time 0:35:07 and MQE 0.55741.

An example of limit values achieved for delay $L = 5\%$ is possible to see in Fig. 5 where we can see the methods *Inc* and *Dec* with step $\zeta = 0.1\%, 0.01\%$

and 0.005%. The step ζ value has a major impact on the overall result, because it determines the time when the above method reaches the maximum permitted delay or vice versa, when they reach the minimum delay.

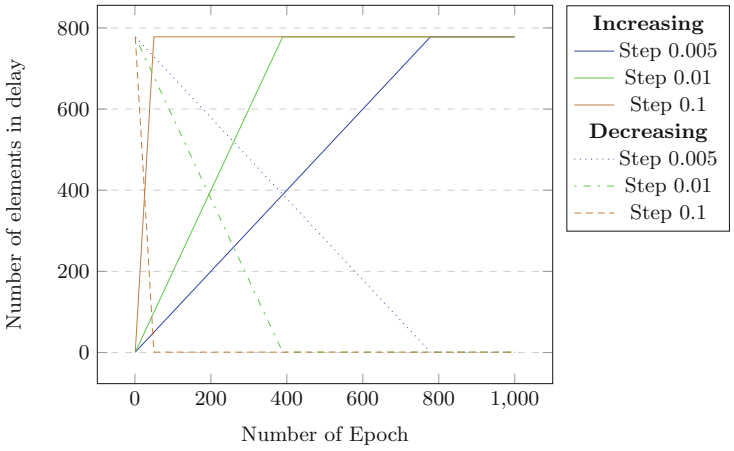


Fig. 5. Example of achievement delay 5%

6 Conclusion

The experiments have shown the possibility of speeding up the computation of actualizing the weights while maintaining the sufficient quality of the final neural network. Speeding up the calculation of the SOM algorithm is based on updating the weights after several (delay L) input vectors. It is similar to Batch SOM, which updates the weights after one epoch. The actualization process for variant *Dec* calculates the values of the weight roughly at the beginning and the next calculation in this variant leads to a more accurate calculation of the weights using the decreasing value of the delay. Overall, the best results are achieved for the variant *Dec* with the smallest test delay ($L = 5\%$) and a mean step ($\zeta = 0.01\%$). This variant is quickly approaching the standard SOM with weight actualization after each input vector. With the initial actualization for the smallest test number of delays, this variant *Dec* is faster than the standard SOM (*Dec* for cores = 16, $L = 5\%$, $\zeta = 0.01\%$ takes 14:19:45 and the standard SOM for 16 cores takes 32:10:30). Further acceleration is due to the massive parallelization, when the best time is achieved for 512 cores (0:35:07). Even faster is the variant with $\zeta = 0.005\%$, but the MQE of this variant is twice as big and therefore less accurate.

Acknowledgments. This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development

Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033), supported by the internal grant agency of VŠB Technical University of Ostrava, Czech Republic, under the project no. SP2015/114 "HPC Usage for Analysis of Uncertain Time Series" and also this work was supported by SGS, VSB - Technical University of Ostrava, Czech Republic, under the grant No. SP2015/146 "Parallel processing of Big Data 2".

References

1. Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 217–235. Springer, Heidelberg (1998)
2. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing inferace. MIT Press (1999)
3. Kohonen, T.: Self-Organization and Associative Memory. Springer Series in Information Sciences, vol. 8, 3rd edn. Springer, Heidelberg (1989)
4. Kohonen, T.: Self Organizing Maps, 3rd edn. Springer-Verlag (2001)
5. Lawrence, R., Almasi, G., Rushmeier, H.: A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. Data Mining and Knowledge Discovery **3**(2), 171–195 (1999)
6. Martinovič, J., Slaninová, K., Vojáček, L., Draždilová, P., Dvorský, J., Vondrák, I.: Effective Clustering Algorithm for High-Dimensional Sparse Data based on SOM. Neural Network World **23**(2), 131–147 (2013)
7. Slaninová, K., Martinovič, J., Novosád, T., Dráždilová, P., Vojáček, L., Snášel, V.: Web site community analysis based on suffix tree and clustering algorithm. In: Proceedings - 2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Workshops, WI-IAT Workshops 2011, pp. 110–113 (2011)
8. Vojáček, L., Dvorský, J., Slaninová, K., Martinovič, J.: Scalable parallel som learning for web user profiles. In: International Conference on Intelligent Systems Design and Applications, ISDA, pp. 283–288 (2014)
9. Wu, C.H., Hodges, R.E., Wang, C.J.: Parallelizing the self-organizing feature map on multiprocessor systems. Parallel Computing **17**(6–7), 821–832 (1991)