Neural Processing Letters **20:** 199–211, 2004.

The Evolving Tree—A Novel Self-Organizing Network for Data Analysis

JUSSI PAKKANEN, JUKKA IIVARINEN and ERKKI OJA
Laboratory of Computer and Information Science, Helsinki University of Technology, P.O. Box 5400, FIN-02015 HUT, Finland. e-mail: jussi.pakkanen@hut.fi

Abstract. The Self-Organizing Map (SOM) is one of the best known and most popular neural network-based data analysis tools. Many variants of the SOM have been proposed, like the Neural Gas by Martinetz and Schulten, the Growing Cell Structures by Fritzke, and the Tree-Structured SOM by Koikkalainen and Oja. The purpose of such variants is either to make a more flexible topology, suitable for complex data analysis problems or to reduce the computational requirements of the SOM, especially the time-consuming search for the best-matching unit in large maps. We propose here a new variant called the Evolving Tree which tries to combine both of these advantages. The nodes are arranged in a tree topology that is allowed to grow when any given branch receives a lot of hits from the training vectors. The search for the best matching unit and its neighbors is conducted along the tree and is therefore very efficient. A comparison experiment with high dimensional real world data shows that the performance of the proposed method is better than some classical variants of SOM.

Key words. self-organization, self-organizing map, tree-structured neural network, image classification.

1. Introduction

The Self-Organizing Map or SOM [1] is a very well-known data analysis tool for tasks like data visualization and clustering. However, some inherent properties of the SOM may be too restrictive in certain applications. In this paper we try to work around these problems by examining the Evolving Tree, a novel neural network system developed by us. It was first introduced in [2].

One of the drawbacks of the SOM is that the user must select the map size in advance. This may lead to many experiments with different sized maps, trying to obtain the optimal result. This is very time consuming. A problem with large SOMs is that when the map size is increased, the time it takes to do any operations on the map increases linearly. Training and using these large maps may again be quite slow.

Several classical approaches have been suggested to circumvent these problems. One approach is to start with a small SOM and add nodes to it during training. Blackmore and Miikkulainen [3] propose *incremental grid growing*. In their system a rectangular grid is used. Neighbors that are close to each other in the data space are connected, whereas very distant ones are not. New map nodes are added at

locations that poorly represent the data space. This approach allows the grid to fit the data better.

Fritzke's *Growing Cell Structure* [4] abandons the symmetric grid structure altogether. Instead his system has a group of hypertetrahedrons. During training new hypertetrahedrons are grown from the old ones. The system also removes harmful hypertetrahedrons for greater flexibility. Related approaches have been presented by Martinez and Schulten [5] and Bruske and Sommer [6].

Growing the size of the grid dynamically is an intuitive and good principle. However it does not solve the other problem mentioned above. When the grid size becomes very large, finding the best matching unit takes longer and longer. Larger maps require some sort of additional data structures that make the search faster.

One possible solution is the *Tree-structured Self-Organizing Map* (TS-SOM) by Koikkalainen and Oja [7]. It has several different sized SOMs arranged in a pyramidal shape. First the topmost SOM is trained in the normal way. Then the next layer is trained, but the algorithm for finding the best matching unit is slightly different. Every node in the first SOM has children in the second SOM. First the best matching unit of the first SOM is found. Then only the children of that node are searched to find the best matching unit of the second SOM. The rest of the training is just like in a regular SOM. More layers can be added in a similar fashion. This structure reduces the amount of operations needed from $O(n)$ to $O(\log n)$. Using the TS-SOM, it has been possible to use and train very large maps, e.g. in an image database application [8].

An ideal system could combine grid growing and hierarchical search. However, there are not many systems that do this. One example is the *Growing Hierarchical Self-Organizing Map* by Dittenbach *et al.* [9]. The system starts with a simple SOM and adds nodes to it as necessary. When a certain criterion is reached, a node is expanded. This means that a new SOM is placed below the node. The training vectors that would map to the parent node are instead used to train the new SOM.

In this paper, we propose and analyze a new self-organizing system called the Evolving Tree. The basic principle was suggested by one of the authors in [2]. In section 2, the Evolving Tree algorithm and architecture are presented. Sections 3 and 4 give two experiments, one artificial, one using real-world data in an image database application. Section 5 has some discussion on the system and some possible future work.

2. Overview of the Evolving Tree

2.1. Architecture and training of the evolving tree

The Evolving Tree has nodes with prototype vectors, just like the SOM. Let us index the nodes by i and let $\mathbf{w}_i \in \mathcal{R}^n$ denote the prototype or weight vector of node i. In addition to that, every node in the network has a counter b_i, giving the number of times it has been the best matching unit during training. Such a counter

has been used earlier in competitive learning algorithms; it was already included in the "conscience" mechanism of Desieno [10] and is also present in Fritzke's Growing Cell Structure [4].

The Evolving Tree algorithm starts by taking a single node and placing it at a suitable place in the data space. An obvious choice is the center of mass of the data cloud. Then we *split* the node. This means that we create a pre-determined amount of new nodes and mark them as the children of the split node. The weight vectors are initialized to that of the parent node, and for the next training vector, the BMU is randomly chosen among the children nodes in a manner described in the next chapter. In this way, the children weights separate from each other. We now have a tree structure with some amount of leaf nodes and one trunk node. All further manipulation operations are only done on the leaf nodes. Once created, a trunk node remains totally static, its only task is to maintain connections between other trunk and leaf nodes in the tree. Once a leaf node reaches a *splitting threshold* θ, having been the BMU for a given number of times, it is again split and so on. Thus a tree is formed recursively in the training algorithm.

Now we come to the novel parts of the algorithm: how to find the BMU in a tree and how to train the tree structure. To illustrate this process we now assume to have a larger tree structure, which can be seen in Figure 1. In this example every trunk node has two children. Finding the BMU is a top–down process. We start with the root node. Then we examine its children and find the node whose prototype vector is closest to the training vector. If that node is a leaf node, then it is the best matching unit. If not, its children are examined in turn and the closest one of them is chosen. This is repeated until a leaf node is found. Thus the Evolving Tree's trunk nodes work as a hierarchical search tree for the leaf nodes. This method of finding the BMU is very similar to the one used in TS-SOM.

When the BMU has been determined, it is time to update the prototype vectors of the BMU, but also those of its neighbors, towards the training vector. The Evolving Tree uses the Kohonen learning rule to update node weight $\mathbf{w}_i(t)$ towards the training vector $\mathrm{x}(t)$ [3]:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + h_{ci}(t)[\mathbf{x}(t) - \mathbf{w}_i(t)]. \tag{1}$$

Here c is the index of the BMU and $h_{ci}(t)$ defines the neighborhood function. These are identical to their counterparts in the regular SOM algorithm. The counter b_c of the BMU is also incremented according to the following formula

$$b_c(t+1) = b_c(t) + 1 \tag{2}$$

If we find after this incrementation that

$$b_c(t+1) = \theta, \tag{3}$$

we split the BMU in the manner described above. This indicates that as the training continues, the tree keeps growing all the time. The speed of growth can be

controlled by the θ parameter. If continuous growth is unwanted, it could be fixed by e.g. zeroing the counters between epochs. This is not currently implemented, because we have had no need for this functionality.

A common choice for the neighborhood is the gaussian neighborhood

$$h_{ci}(t) = \alpha(t) \exp\left(\frac{\| \mathbf{r}_c - \mathbf{r}_i \|^2}{2\sigma^2(t)}\right). \tag{4}$$

The vectors $\mathbf{r}_c$ and$\mathbf{r}_i$ give the locations of nodes c and i on the SOM grid, the parameter $\alpha(t)$ defines the learning rate, and $\sigma(t)$ gives the width of the gaussian kernel.

In the Evolving Tree parameters $\alpha(t)$ and $\sigma(t)$ work just like in the SOM. However, calculating the norm $\| \mathbf{r}_c - \mathbf{r}_i \|$ is not as simple. On the SOM it is just the grid distance between the node to be updated and the best matching unit. This is easy to calculate, because SOM is symmetric, static and has regular structure. None of these properties necessarily applies for the Evolving Tree.

There is, however, a simple way to calculate an equivalent distance value in the tree. The principle is shown in Figure 1. In a tree structure, there is always the shortest path between any given two nodes. The minimal distances from the BMU node to all the other nodes in the tree can be computed using the following efficient algorithm. The basic idea is to calculate how many 'hops' must be done to get from the BMU to the desired node along this shortest path. The exact distance is the amount of hops minus one. One is subtracted because we are only interested in distances between leaf nodes. The closest possible leaf nodes have a common parent, so it takes two hops to get from one leaf to another. We want these 'closest neighbors' to have a distance of one, so we subtract one from all distances to retain consistency. In the example shown in Figure 1 it takes five hops to get from node A to node B, so the tree distance between these nodes is four.

It should be noted that our choice of distance is unique and sensible. The way the Evolving Tree is formed ensures that there is one, and only one, path between any two given leaf nodes (and also trunk nodes). Therefore the distance function is unique and symmetric. Since the path is unique, it must also be the shortest

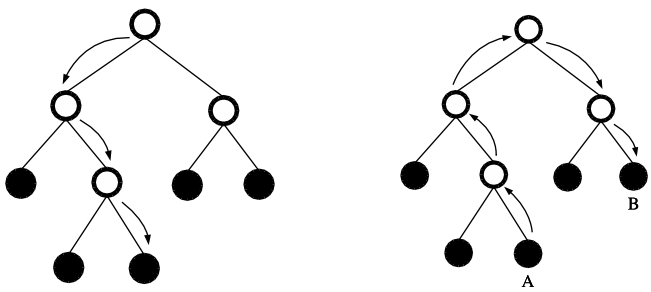

Figure 1. Fundamental operations of the Evolving Tree. The left image shows how the BMU is found. The right image demonstrates how the tree distance between two nodes is calculated.

path. These properties prove that the function is sensible mathematically. It is also suitable intuitively since we can assume that different branches of the tree grow to different areas of the data space. Therefore distances inside a branch are smaller than between different branches.

2.2. RELATION TO OTHER TREE-SHAPED SYSTEMS

The Evolving Tree can also be seen as a tree-structured index to an n-dimensional metric space. Making fast searches in metric spaces is a problem shared by many disciplines, such as databases and computer graphics. This has caused a lot of unnecessary re-inventing of the wheel. An overview on the subject can be found in [11].

To our knowledge the Evolving Tree is not just a reformulation of some earlier data analysis method. The main reason is the self organization property, which has not been used much in classifiers. Conversely combinations of search structures and SOMs (such as the ones listed in the introduction) differ noticeably from the Evolving Tree. Even the most similar system, the TS-SOM, is very different, since it adds regular SOM layers one at a time. In contrast the Evolving Tree grows much more freely, one node at a time.

3. Time Behaviour Visualization Experiment

Since the Evolving Tree is a new system, it is important to understand how it behaves during training. To do this we created an easily visualizable two-dimensional data set. First we created a picture that has the text 'Evolving Tree' written on it. This can be seen in Figure 2. Then we used this picture as a probability distribution. The area under the letters has a uniform probability while the areas where there is no text have zero probability. We drew one thousand two-dimensional data vectors from this distribution and used them as the training set $x(t)$, $t = 0, \ldots, 999$.

Any leaf node was split when its counter reached the threshold $\theta = 60$, i.e., it had been the BMU 60 times. Then the node was always divided into four new nodes. Ten epochs were used in the training. Within each epoch consisting of all the training vectors, the vectors were randomly shuffled in order to prevent nodes from having a strong bias. This is especially important in the early stages of training, when the nodes are BMUs relatively often.

Figure 2 demonstrates how the Evolving Tree changes as it is being trained. The first image shows, as was mentioned above, the uniform distribution where the samples were drawn from. The next one shows the locations (weight vectors) of the leaf nodes after only a few hundred training vectors have been used. We can see that the tree has been split several times and that the nodes have found the general shape of the data cloud. It can be immediately seen that these prototype vectors can not be used for any real data analysis work. The way they partition the data space is simply too coarse.

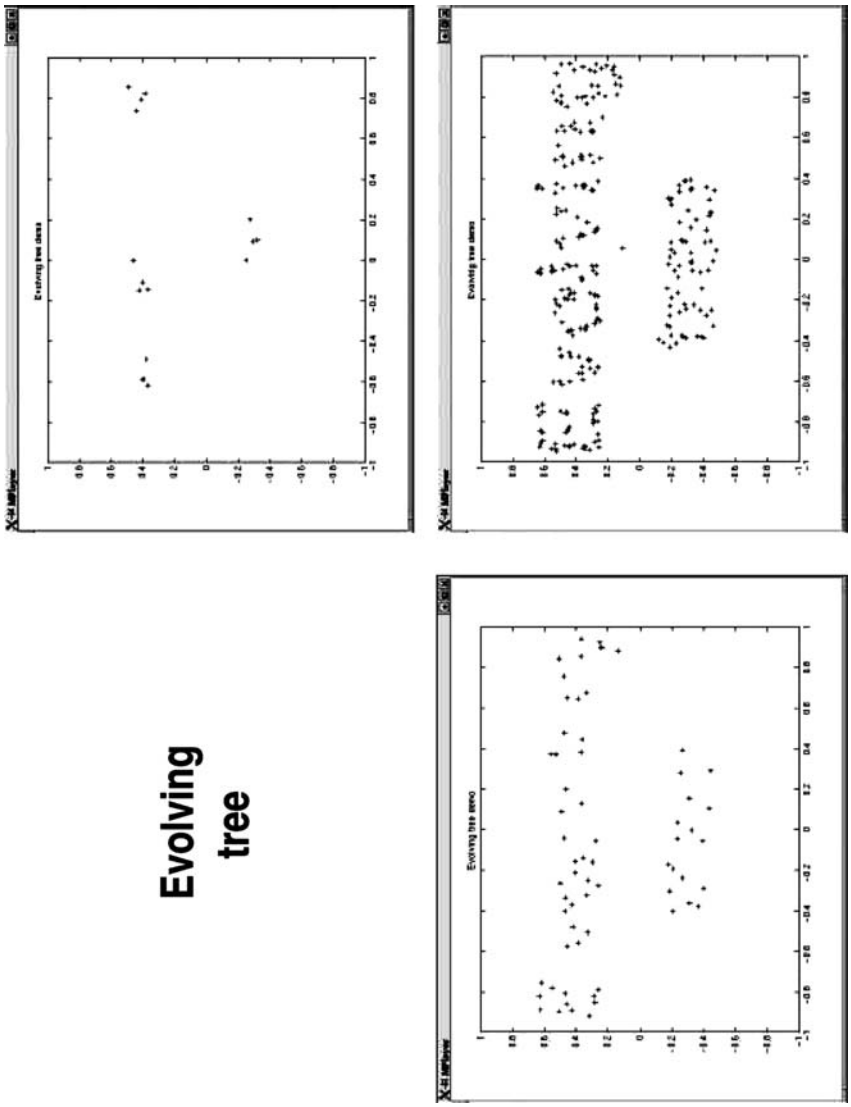

Figure 2. Training the Evolving Tree. The first image shows the data distribution, and the other three show the state of the tree as the training progresses. The dots mark the locations of the leaf nodes.

As the training continues, more and more leaf nodes appear. The third picture shows that the tree has adapted itself a lot better to the data. The overall shape is quite good and even some individual letters can be seen. At this point the tree could even be used for data analysis, even though the overall shape is a bit coarse.

After 10 epochs we have the structure shown in the last picture of Figure 2. The Evolving Tree has adapted itself very strongly to the data. So much, in fact, that you can read the original text just by looking at the leaf node distribution.

3.1. OBSERVATIONS

Several interesting properties of the Evolving Tree can be seen in the images in Figure 2. The first thing is that the leaf nodes' prototype vectors are scattered quite evenly on the text. This is very desirable, as the original distribution was uniform. There are some places, such as the bottom of the letter 'E', where the nodes are denser. This is quite expected, given the small size of the data set and the nature of the algorithm.

The images also verify what we assumed earlier: different branches correspond to different areas of the data space. The first iteration rounds divide the space roughly and then specialize in those areas. This is done by spreading a few nodes roughly on the area of the parent branch and then splitting the nodes for finer modeling. These split nodes again work in the same way. This becomes especially clear when looking at a video clip of the training process. Lacking that, these images should give an overall feeling of how the tree grows with time.

One more thing to notice is that there are very few nodes that are not inside some letter. One example can be seen below the letters 'l' and 'v'. This is a very positive property for any data analysis tool. It means that the system focuses only on those areas of data space that have data vectors. It should be noted that in this example we wanted to get a good visualization, so the nodes were split very aggressively, meaning a node was split after it had been the BMU only 60 times. This is the reason the above mentioned node gets 'stranded in the void' between data areas. If we raise the threshold θ and thus wait longer until splitting a node, all nodes end up on some letter. Had we trained a regular SOM with the same data, there would have been several nodes in the void between letters. This is due to the rigid structure of the SOM grid.

4. Real World Data Set Experiment

A comparison between a regular SOM, a supervised SOM [1], the Tree Structured SOM [7] and our Evolving Tree algorithm was conducted using a large real world data set in order to see how the Evolving Tree compares with the other methods.

For the SOM and the Supervised SOM we used a well known software package called the *SOM Toolbox*.[1] The system has heuristic functions which automatically

[1] http://www.cis.hut.fi/projects/somtoolbox.

deduce correct training parameters. The most important parameter is the map size which was chosen to be 19×9. For TS-SOM training we utilized the implementation of the PicSOM project [12]. TS-SOMs had two layers of size 4×4 and 16×16. Since the Evolving Tree is a new system, choosing the parameters was not a straightforward task. There are two crucial parameters. The first is the *splitting threshold* θ, indicating how many times a unit must be the BMU before splitting. The other one is the *splitting factor*, which tells how many children a node will have. Experiments indicated that values of 50 and 3, respectively, gave good results. Eight epochs were used to train the trees.

As the SOM, the TS-SOM, and the Evolving Tree are unsupervised methods, measuring and comparing their performances is not simple. For this experiment, we used a set of classified images as our training material and looked at the classification errors. None of these methods can exceed a proper classifier trained with supervised algorithms, but the comparison between the unsupervised methods may still be fair and informative. All tests used 10-fold cross-validation to obtain unbiased classification results.

4.1. DEFECT IMAGE DATABASE

The experiments were conducted with a set of paper defect (fault) images. Some examples are shown in Figure 3. All the images were obtained from a real, online visual quality control system on a running paper machine. As can be seen from Figure 3, the images have different kinds of defects and their sizes vary according to the size of a defect. Classification of defects is based on the cause and type of a defect, and different classes can therefore contain images that are visually similar in many aspects. The images have 256 gray levels and they have been automatically

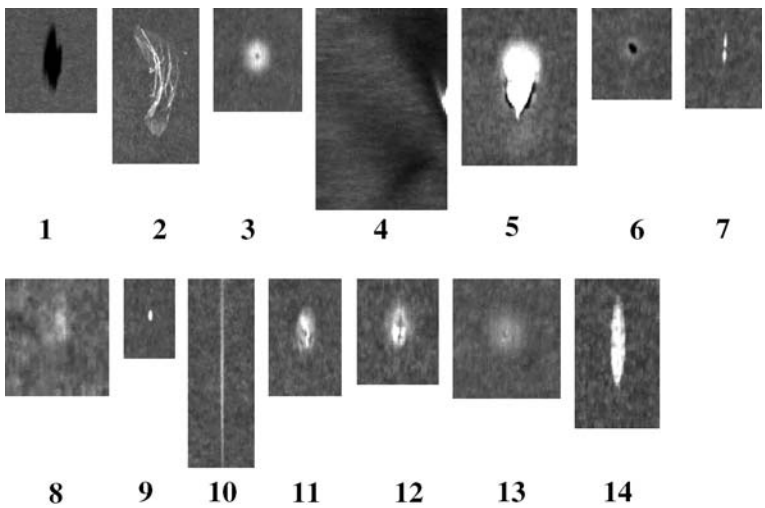

Figure 3. Example images from each class of the paper defect database. Courtesy of ABB oy, Finland.

segmented. Thus each image has a gray level image and a binary segmentation mask image that indicates defect areas in the image.

There are 1308 defect images in our data set. They have been manually preclassified into 14 different classes. Most classes had approximately one hundred images, three classes had 63 images and class number 12 had only 27 images. Example images from each class are depicted in Figure 3. It shows how many classes are very similar, especially classes 11 and 12. The images are in fact so hard to classify that even a professional paper maker might not be able to classify them correctly.

4.2. FEATURE EXTRACTION BY MPEG-7 DESCRIPTORS

Before trying to automatically classify such images, they must be reduced to numerical vector representations by suitable feature extraction. For this, the MPEG-7 standard, ISO/IEC 15938, formally named 'Multimedia Content Description Interface' [13–15], has been found especially suitable. It provides standardized descriptions of streamed or stored images or video, to be used in searching, identifying, filtering and browsing images or video in various applications. The standard defines several still image descriptors. Based on our earlier results with paper defect images in [16], the following three descriptors were clearly ranked as the best ones and thus they are also used in these experiments:

- *Color structure (CS)* slides a structuring element over the image. The numbers of positions where the element contains each particular color are stored and used as a descriptor.
- *Edge histogram (EH)* calculates the amount of vertical, horizontal, 45°, 135° and non-directional edges in 16 sub-images of the picture, resulting in a total of 80 histogram bins.
- *Homogeneous texture (HT)* filters the image with a bank of orientation and scale tuned filters that are modeled using Gabor functions. The first and second moments of the energy in the frequency domain in the corresponding sub-bands are then used as the components of the texture descriptor.

4.3. RESULTS

The classification results, averaged over the 10-fold cross-validation experiments, can be seen in Figure 4. The first figure shows the classification results averaged over the three features. The overall results of the Evolving Tree are noticeably better than the SOM percentages. The only significant drop-off is in class 4. The Evolving Tree makes up for this poor performance by maintaining a steady classification rate on the very difficult classes 11 and 12. This kind of good performance on the hard portions of the data space is very desirable.

The second graph in Figure 4 gives a better overall view of the classification results. It clearly shows that the Evolving Tree performs slightly better than the

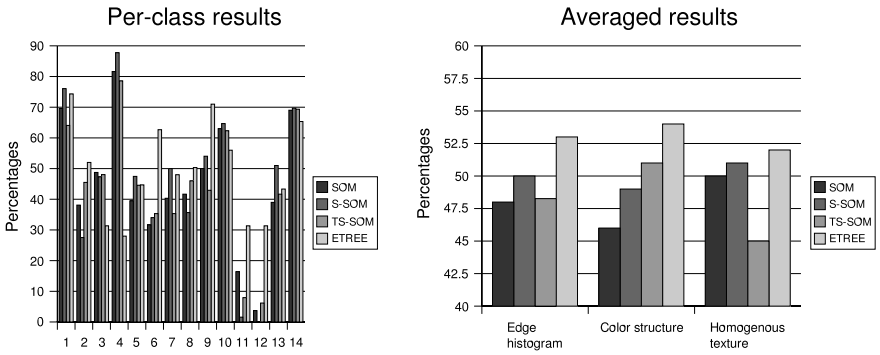

Figure 4. The left graph shows the results averaged over features while the left one is averaged over classes. Note the difference in scale.

supervised SOM, which is better than the regular SOM. TS-SOM's performance varies quite a lot, but being roughly the same as regular SOMs.

Another important aspect is the complexity of the data structures used in the training. The SOMs used maps of size 19×9, having a total of 171 nodes. The Evolving Tree's size is not constant. Due to the splitting rule, the amount of leaf nodes depend on many different things, including the ordering of the training vectors. In these experiments the trees had approximately 240 leaf nodes.

This quite large difference in complexity may seem like a lot, but it can be explained by the architectural differences between the SOM and the Evolving Tree. The trunk nodes of the Evolving Tree maintain a very efficient search tree to the leaf nodes at all times. Searches on the tree can be performed in $O(\log n)$ time compared to $O(n)$ of the regular SOM. This structure makes the Evolving Tree handle complexity very well. The TS-SOM also has a $O(\log n)$ search structure, but it has noticeably worse classification performance than the Evolving Tree. This poor performance is amplified by the fact that the TS-SOM has $16 \times 16 = 256$ nodes in its lowest layer making it more complex than the Evolving Tree.

One of the good features of the Evolving Tree is its efficiency, so comparing the run times of the different algorithms is in order. Unfortunately we cannot give CPU times that would be objective. The reason for this is the different programming environments used. The Evolving Tree is implemented as a very light weight C++ program. In contrast the SOM and S-SOM implementations are made in Matlab. Given the nature of the SOM calculations, the Matlab platform exerts a heavy penalty on efficiency. Testing subjectively, the Evolving Tree was the fastest while the SOM and S-SOM were the slowest, having an execution time several times longer. This is consistent with the complexity calculations above. Since these tests we have done optimizations on the Evolving Tree program, which make it over 10 times faster.

5. Discussion and Future Work

A brief analysis of the random process involved in the Evolving Tree training can be done as follows: assume a node has reached the threshold, has been frozen, and n children have been spawned. Now the Voronoi region of the parent node, defined as the set of training vectors for which it is the BMU, is always the union of the Voronoi regions of the child nodes. The areas of the children's Voronoi regions are roughly $\frac{1}{n}$ of that of the parent node. Thus also the variance of the training vectors falling into these Voronoi regions is $\frac{1}{n}$. Going deeper into the tree, the variances very soon become small and the leave nodes represent very accurately a small part of the training set.

Note also that the size of the tree (the number of nodes) is determined by the length of training. Starting from one root node, after θ steps of online training it is split into n child nodes. Assuming that each node has equal probability of being the BMU, it takes on the average $n\theta$ steps until a new generation of n^2 children is spawned. The next generation of n^3 children is formed after additional $n^2\theta$ steps, etc. The process of splitting can of course be stopped by raising the threshold to a very large value; then the tree architecture does not change any more, but the node weights keep on adjusting to the training data.

Our simple node splitting rule is not the only way to decide when to split nodes. Arbitrarily complex decision rules could be used for this task. These rules could, for example, examine the densities of data and prototype vectors in the immediate vicinity of the BMU. Development of these rules is one of the key tasks in our future work. But in this paper we have chosen to use the simple threshold rule for clarity and because it produces quite good results.

Another interesting problem is the initialization of the new child nodes. In this paper we simple put all of them over the parent node and then randomly choose the BMU among them for the next training vector. A better approach could be to initialize them along a two-dimensional subspace spanned by the two principal directions (or eigenvectors) of the training vectors of the parent node's Voronoi region. This kind of ordered initialization is commonly used in training of normal SOMs, but implementing it for the Evolving Tree is not straightforward. However, it should speed up training, give better results, and especially make training more repeatable.

The way the Evolving Tree is constructed does unfortunately not guarantee that the BMU found with the tree search is the true BMU among all the leaf nodes. We have done some preliminary experiments and found that in over 70% of searches the BMUs are the same. However their tree distance is always very small, usually around three, but almost always less than 8. Interestingly if we use the 'best' BMU instead of the one found using the tree search, the classification results are almost exactly the same. This is an interesting result, which warrants further study.

6. Conclusions

In this paper we have presented and analyzed the Evolving Tree, a new variant of self-organizing neural networks. It is a growing tree structured topology of competitive—co-operative neural nodes. Growing is based on the frequency of being the best-matching unit. Once this frequency exceeds a threshold for a node, it is split into a number of child nodes. Only the leaf nodes learn according to the SOM learning rule. Such growing networks have been proposed earlier, notably by Fritzke in his Growing Cell Structures (GCS) [4]. The difference of the Evolving Tree is that, while GCS is based on hypertetrahedrons and the network topology has a fixed dimension, the neighborhood in the Evolving Tree is defined along a tree. Contrary to the Tree-Structured SOM of Koikkalainen and Oja [7], which also has a hierarchical structure, the Evolving Tree does not form regular SOM layers, but each branch is free to evolve independently. This flexibility allows the Evolving Tree to adapt to data more efficiently, leading to better performance.

Experiments with artificial and real world data showed that the Evolving Tree is noticeably better in performance than the basic SOM and the TS-SOM. It has computational complexity relative to the TS-SOM, but the topology is more flexible. The Evolving Tree may thus be a very suitable self-organizing network for tasks in which quantization and classification is more important than visualization, especially when dealing with huge data sets.

Acknowledgments

The authors would like to thank Mr J. Rauhamaa of ABB Oy for his help and comments. The financial support of the Technology Development Centre of Finland (TEKES's grant 40120/03) and ABB oy is gratefully acknowledged.

References

1. Kohonen, T.: *Self-Organizing Maps*. Berlin: Springer-Verlag, (1995).
2. Pakkanen, J.: The Evolving Tree, a new kind of self-organizing neural network. In: *Proceedings of the Workshop on Self-Organizing Maps '03*, Kitakyushu, Japan, pp. 311–316, 2003.
3. Blackmore, J. and Miikkulainen, R.: Incremental grid growing: encoding high-dimensional structure into a two-dimensional feature map. In: *Proceedings of the IEEE International Conference on Neural Networks,* vol. 1. pp. 450–455, 1993.
4. Fritzke, B.: Growing cell structures—a self-organizing network for unsupervised and supervised learning. *Neural Networks* **7**(9) (1994), 1441–1460.
5. Martinez, T. and Schulten, K.: A "Neural-Gas" Network Learns Topologies. In: T. Kohonen, K. Mäkisara, O. Simula and J. Kangas (eds.), *Artificial Neural Networks*, vol.1. Amsterdam, pp. 297–402, 1991.
6. Bruske, J. and Sommer, G.: Dynamic cell structure learns perfectly topology preserving map. *Neural Computation* **7**(4) (1997), 845–865.

7. Koikkalainen, P. and Oja, E.: Self-Organizing hierarchical feature maps. In: *Proceedings of 1990 International Joint Conference on Neural Networks*, vol. II. San Diego, CA, pp. 279–284, 1990.
8. Laaksonen, J., Koskela, M., and Oja, E.: PicSOM— self-organizing image retrieval with MPEG-7 content descriptors. *IEEE Transactions on Neural Networks* **13**(4) (2002), 841–853.
9. Dittenbach, M., Rauber, A., and Merkl, D.: Recent advances with the growing hierarchical self-organizing map. In: *Proceedings of the 3rd Workshop on Self-Organizing Maps*, Lincoln, England, pp. 140–145, 2001. http: //www.ifs.tuwien.ac.at.
10. Desieno, D.: Adding a conscience to competitive learning. In: *Proceedings of the International Conference on Neural Networks*, vol.I, New York, pp. 117–124, 1998.
11. Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquin J. L.: Searching in metric spaces. *ACM Computing Surveys* **33**(1) (2001), 273–321.
12. Laaksonen, J., Koskela, M., Laakso, S., and Oja, E.: PicSOM—Content-based image retrieval with self-organizing maps. *Pattern Recognition Letters* **21**(13–14) (2000), 1199–1207.
13. Manjunath, B. S., Ohm, J.-R., Vasudevan, V. V., and Yamada, A.: Color and texture descriptors. *IEEE Transactions on Circuits and Systems for Video Technology* **11**(6) (2001).
14. MPEG-7: *MPEG-7 Multimedia Content Description Interface — Part 3 Visual*. ISO/IEC JTC1/SC29/WG11 W3703 2001.
15. MPEG-7: *MPEG-7 Visual Part of the eXperimentation Model (version 9.0)*. ISO/IEC JTC1/SC29/WG11 N3914, 2001.
16. Pakkanen, J., Ilvesmäki, A. and Iivarinen, J.: 'Defect image classification and retrieval with MPEG-7 descriptors'. In: J. Bigun and T. Gustavsson (eds.), *Proceedings of the 13th Scandinavian Conference on Image Analysis*, Göteborg, Sweden, pp. 349–355, 2003.