A general framework for unsupervised processing of structured data

Barbara Hammer [1]

Research group LNM, Department of Mathematics/Computer Science, University of Osnabrück, Germany

Alessio Micheli

Dipartimento di Informatica, Università di Pisa, Pisa, Italia

Marc Strickert

Research group LNM, Department of Mathematics/Computer Science, University of Osnabrück, Germany

Alessandro Sperduti

Dipartimento di Matematica Pura ed Applicata, Università degli Studi di Padova, Padova, Italia

Abstract

Self-organization constitutes an important paradigm in machine learning with successful applications e.g. in data- and web-mining. Most approaches, however, have been proposed for processing data contained in a fixed and finite dimensional vector space. In this article, we will focus on extensions to more general data structures like sequences and tree structures. Various modifications of the standard self-organizing map (SOM) to sequences or tree structures have been proposed in the literature some of which are the temporal Kohonen map, the recursive SOM, and SOM for structured data. These methods enhance the standard SOM by utilizing recursive connections. We define a general recursive dynamic in this article which provides recursive processing of complex data structures by recursive computation of internal representations for the given context. The above mentioned mechanisms of SOMs for structures are special cases of the proposed general dynamic. Furthermore, the dynamic covers the supervised case of recurrent and recursive networks. The general framework offers a uniform notation for training mechanisms such as Hebbian learning. Moreover, the transfer of computational alternatives such as vector quantization or the neural gas algorithm to structure processing networks can be easily achieved. One can formulate general cost functions corresponding to vector quantization, neural gas, and a modification of SOM. The cost functions can be compared to Hebbian learning which can be interpreted as an approximation of a stochastic gradient descent. For comparison, we derive the exact gradients for general cost functions.

Preprint submitted to Elsevier Science 27 May 2003

Key words: Self organizing map, Kohonen map, recurrent networks, SOM for structured data

1 Introduction

Neural networks constitute a particularly successful approach in machine learning which allows to learn an unknown regularity for a given set of training examples. They can deal with supervised or unsupervised learning tasks; hence outputs or classes for the data points might be available and the network has to learn how to assign given input data appropriately to the correct class in the supervised case. Alternatively, in the unsupervised case, no prior information about a valid separation into classes is known and the network has to extract useful information and reasonable classes from data by itself. Naturally, the latter task is more difficult because the notion of 'useful information' depends on the context. Results are often hard to evaluate automatically, and they must be validated by experts in the respective field. Nevertheless, the task of unsupervised information processing occurs in many areas of application for which explicit teacher information is not yet available: data- and Web-mining, bioinformatics, or text categorization, to name just a few topics. In the context of neural networks, most approaches for supervised or unsupervised learning deal with finite dimensional vectors as inputs. For many areas of interest such as time-series prediction, speech processing, bioinformatics, chemistry, or theorem proving, data are given by sequences, trees, or graphs. Hence data require appropriate preprocessing in these cases such that important features are extracted and represented in a simple vector representation. Preprocessing is usually domain dependent and time consuming. Moreover, loss of information is often inevitable. Hence, effort has been done to derive neural methods which can deal with structured data directly.

In the supervised scenario, various successful approaches have been developed: Supervised recurrent neural networks constitute a well established approach for modeling sequential data e.g. for language processing or time series prediction [16,17]. They can naturally be generalized to so-called recursive networks such that more complex data structures, tree structures and directed acyclic graphs can be dealt with [14,47]. Since symbolic terms possess a tree-representation, this generalization has successfully been applied in various areas where symbolic or hybrid data structures arise such as theorem proving, chemistry, image processing, or natural

Email address: `hammer@informatik.uni-osnabrueck.de` (Barbara Hammer).

[1] The research has been done while the first author was visiting the University of Pisa. She would like to thank the groups of Padua, Pisa, and Siena for their warm hospitality during her stay.

language processing [1,2,9,13]. The training method for recursive networks is a straightforward generalization of standard backpropagation through time [46,47]. Moreover, important theoretical investigations from the field of feedforward and recurrent neural networks have been transferred to recursive networks [13,15,21].

Unsupervised learning as alternative important paradigm for neural networks has been successfully applied in data mining and visualization (see [31,42]). Since additional structural information is often available in possible applications of self-organizing maps (SOMs), a transfer of standard unsupervised learning methods to sequences and more complex tree structures would be valuable. Several approaches extend SOM to sequences: SOM constitutes a metric based approach, therefore it can be applied directly to structured data if data comparison is defined and a notion of adaptation within the data space can be found. This has been proposed e.g. in [18,29,48]. Various approaches alternatively extend SOM by recurrent dynamics such as leaky integrators or more general recurrent connections which allow the recursive processing of sequences. Examples are the temporal Kohonen map (TKM) [5], the recursive SOM (RecSOM) [50–52], or the approaches proposed in [11,27,28,33]. The SOM for structured data (SOMSD) [19,20,46] constitutes a recursive mechanism capable of processing tree structured data, and thus also sequences, in an unsupervised way. Alternative models for unsupervised time series processing use, for example, hierarchical network architectures. An overview of important models can be found e.g. in [3].

We will here focus on models based on recursive dynamics for structured data and we will derive a generic formulation of recursive self-organizing maps. We will propose a general framework which transfers the idea of recursive processing of complex data for supervised recurrent and recursive networks to the unsupervised scenario. This general framework covers TKM, RecSOM, SOMSD, and the standard SOM. The methods share the basic recursive dynamic but they differ in the way in which structures are internally represented by the neural map. TKM, RecSOM, SOMSD, and the standard SOM can be obtained by an appropriate choice of internal representations in the general framework. Moreover, the dynamic of supervised recurrent and recursive networks can be integrated in the general framework as well. The approaches reported in [11,27,33] can be simulated with slight variations of parts of the framework. Hence we obtain a uniform formulation which allows a straightforward investigation of possible learning algorithms and theoretical properties of several important approaches proposed in the literature for SOMs with recurrence.

The reported models are usually trained with Hebbian learning. The general formulation allows to formalize Hebbian learning in a uniform manner and to immediately transfer alternatives like the neural gas algorithm [37] or vector quantization to the existing approaches. For standard vector-based SOM and alternatives like neural gas, Hebbian learning can be (approximately) interpreted as a stochastic gradient descent method on an appropriate error function [25,37,42]. One can uni-

formly formulate analogous cost functions for the general framework for structural self-organizing maps and investigate the connection to Hebbian learning. It turns out that Hebbian learning can be interpreted as an approximation of a gradient mechanism for which contributions of substructures are discarded. The exact gradient mechanism includes recurrent neural network training as a special case, and explicit formulae comparable to backpropagation through time or real time recurrent learning [40] can be derived for the unsupervised case. This gives some hints to the understanding of the dynamics of unsupervised network training and constitutes a first step towards a general theory of unsupervised recurrent and recursive networks.

We will now define the general framework formally and show that SOMs with recursive dynamics as proposed in the literature can be recovered as special cases of the general framework. We show how Hebbian learning can be formulated within this approach. Finally, we relate Hebbian learning to alternative training methods based on energy functions, such that popular methods in the field of unsupervised learning can be directly transferred to this general framework and supervised and unsupervised training methods can be related to each other.

2 Structure processing self-organizing maps

We first clarify a notation: the term 'self-organizing map' used in the literature refers to both, the paradigm of a neural system which learns in a self-organizing fashion, and the specific and very successful self-organizing map which has been proposed by Kohonen [31]. In order to distinguish between these two meanings, we refer to the specific architecture proposed by Kohonen by the shorthand notation SOM. If we speak of self-organization, the general paradigm is referred to. The SOM as proposed by Kohonen is a biologically motivated neural network which learns via Hebbian learning a topological representation of a data distribution from examples. Assume data are taken from the real-vector space $\mathbb{R}^n$ equipped with the Euclidian metric $\|\cdot\|$. The SOM is defined as a set of neurons $N = \{n_1, \dots, n_N\}$ together with a neighborhood structure of the neurons $nh : N \times N \to \mathbb{R}$. This is often determined by a regular lattice structure, i.e. neurons n_i and n_j are direct neighbors with $nh(n_i, n_j) = 1$ if they are directly connected in the lattice. For other neurons, $nh(n_i, n_j)$ reflects the minimum number of direct connections needed to link n_i to n_j. A two-dimensional lattice offers the possibility of easy visualization which is used e.g. in data mining applications [34]. Each neuron n_i is equipped with a weight $w_i \in \mathbb{R}^n$ which represents the corresponding region of the data space. Given a set of training patterns $T = \{a_1, \dots, a_d\}$ in $\mathbb{R}^n$, the weights of the neurons are adapted by Hebbian learning including neighborhood cooperation such that the weights w_i represent the training points T as accurately as possible and the topology of the neurons in the lattice matches the topology induced by the data points. The precise learning rule is very intuitive:

repeat:

choose $a_i \in T$ at random

compute neuron n_{i_0} with minimum distance $\|a_i - w_{i_0}\|^2 = \min_j \|a_i - w_j\|^2$

adapt for all j: $w_j := w_j + \eta(nh(n_j, n_{i_0}))(a_i - w_j)$

where $\eta(nh(n_j, n_{i_0}))$ is a learning rate which is maximum for the winner $n_j = n_{i_0}$ and decreasing for neurons n_j which are not direct neighbors of the winner n_{i_0}. Often, the form $\eta(nh(n_j, n_{i_0})) = \exp(-nh(n_j, n_{i_0}))$ is used, possibly adding constant factors to the term. The incorporation of topology in the learning rule allows to adapt the winner and all neighbors at each training step. After training, the SOM is used with the following dynamic: given a pattern $a \in \mathbb{R}^n$, the map computes the winner, i.e. the neuron with smallest distance $\|a - w_j\|^2$ or its weight, respectively. This allows to identify a new data point with an already learned prototype. Starting from the winning neuron, map traversal reveals similar known data.

Popular alternative self-organizing algorithms are vector quantization (VQ) and the neural gas algorithm (NG) [38]. VQ aims at learning a representation of the data points without topology preservation. Hence no neighborhood structure is given in this case and the learning rule adapts only the winner at each step:

repeat:

choose $a_i \in T$ at random

compute neuron n_{i_0} with minimum distance $\|a_i - w_{i_0}\|^2 = \min_j \|a_i - w_j\|^2$

adapt $w_{i_0} := w_{i_0} + \eta(a_i - w_{i_0})$

where $\eta > 0$ is a fixed learning rate. In order to avoid inappropriate data representation caused by topological defects, NG does not pose any topological constraints on the neural arrangement. Rather, neighborhood is defined posteriorly through training data. The recursive update reads as

repeat:

choose $a_i \in T$ at random

compute all distances $\|a_i - w_j\|^2$

adapt for all j: $w_j := w_j + \eta(rk(i,j))(a_i - w_j)$

where $rk(i,j)$ denotes the rank of neuron n_j according to the distances $\|a_i - w_j\|^2$, i.e. the number of neurons n_k for which $\|a_i - w_k\|^2 < \|a_i - w_j\|^2$ holds. $\eta(rk(i,j))$ is a function with maximum at 0 and decreasing values for larger numbers, e.g. $\eta(rk(i,j)) = \exp(-rk(i,j))$ possibly with additional constant factors. This results in the closest neuron to be adapted most, all other neurons are adapted according

to their distance from the given data point. Hence, the respective order of the neurons with respect to a given training point determines the current neighborhood. Eventually, those neurons which are the neurons closest to at least one data point become neighbored. One can infer a data-adapted though no longer regular lattice after training in this way which preserves the topology of the data space [37,38,49].

There exist various possibilities of extending self-organizing maps such that they can deal with alternative data structures instead of simple vectors in $\mathbb{R}^n$. An interesting line of research deals with the adaptation of self-organizing maps to qualitative variables where the Euclidian metric cannot be used directly [7,8]. In this article, we are particularly interested in complex discrete structures such as sequences and trees. The article [3] provides an overview of self-organizing networks which have been proposed for processing spatio-temporal patterns. Naturally, a very common way of processing structured data with self-organizing mechanisms relies on adequate preprocessing of sequences. They are represented through features in a finite dimensional vector space for which standard pattern recognition methods can be used. A simple way of sequence representation is obtained by a truncated and fixed dimensional time-window of data. Since this method often yields too large dimensions, SOM with the standard Euclidian metric suffers from the curse of dimensionality, and methods which adapt the metric as proposed for example in [23,30,45] are advisable. Hierarchical and adaptive preprocessing methods which involve SOMs at various levels can be found e.g. in the WEBSOM approach for document retrieval [34]. Since self-organizing algorithms can immediately be transformed to arbitrary metrical structures instead of the standard Euclidian metric, one can alternatively define a complex metric adapted to structured data instead of adopting a complex data preprocessing. SOMs equipped with the edit distance constitute one example [18]. Data structures might be contained in a discrete space instead of a real vector space in these approaches. In this case one has to specify additionally how the weights of neurons are adapted. If the edit distance is dealt with, one can perform a limited number of operations which transform the actual weight of the neuron to the given data structure, for example. Thereby, some unification has to be done since the operations and their order need not be unique. Some methods propose the representation of complex data structures in SOM with complex patterns, e.g. the activation of more than one neuron. Activation patterns arise through recursive processing with appropriate decaying and blocking of activations like in SARDNET [28]. Alternative methods within these lines can be found in [3].

We are here interested in approaches which equip the SOM with additional recursive connections which make recursive processing of a given recursive data structure possible. In particular, no prior metric or preprocessing is chosen but the similarity of structures evolves through the recursive comparison of the single parts of the data. Various approaches have been proposed in the literature. Most of them deal with sequences of real vectors as input data. One approach has been proposed for more general tree structures and thus including sequences as a special case. We shortly summarize important recursive dynamics in the following. Thereby,

sequences over $\mathbb{R}^n$ with entries a_1 to a_t are denoted by $[a_1, \ldots, a_t]$. $[]$ denotes the empty sequence. Trees with labels in $\mathbb{R}^n$ are denoted in a prefix notation by $a(t_1, \ldots, t_k)$ where a denotes the label of the root, t_1 to t_k the subtrees. The empty tree is denoted by ξ.

Temporal Kohonen Map

The temporal Kohonen map proposed by Chappell and Taylor [5] extends the SOM by recurrent self-connections of the neurons such that the neurons act as leaky integrators. Given a sequence $[a_1, \ldots, a_t]$, the integrated distance of neuron n_i with weight w_i is computed as

$$d_i(t) = \sum_{j=1}^{t} \alpha \cdot (1-\alpha)^{(t-j)} \cdot \|a_j - w_i\|^2$$

where $\alpha \in (0, 1)$ is a constant which determines the integration of context information when the winner is computed. This formula has the form of a leaky integrator which integrates previous distances of neuron n_i given the entries of the sequence. Hence a neuron becomes the winner if its weight is close to the given data point and, in addition, the exponentially weighted distance of previous entries of the sequence from the weight is small. Obviously, an alternative recursive formalization of the integrated distance $d_i(j)$ of neuron i after the j th time step is given by $d_i(j) = \alpha\|a_j - w_i\|^2 + (1-\alpha)d_i(j-1)$ where $d_i(0) := 0$. Training of the TKM is performed in [5] with Hebbian learning after each time step, i.e. the weights w_i are adapted at each time step with respect to the current input a_j according to the standard SOM update rule. Thereby, the winner is computed as the neuron with the least integrated distance according to the above leaky integration formula.

The recurrent SOM (RSOM) as defined in [33], for example, uses a similar dynamic. However, it integrates the directions of deviations of the weights. Hence the activation $d_i(j)$ which is now an element of $\mathbb{R}^n$ is recursively computed by $d_i(0) = 0$, $d_i(j) = \alpha(a_j - w_i) + (1-\alpha)d_i(j-1)$. The winner is determined as the neuron with smallest integrated distance, i.e. smallest $\|d_i(j)\|^2$. Naturally, this procedure stores more information than the weighted distance of the TKM. Again, Hebbian learning can be used at each time step, like in TKM. In [33] an alternative direct training method is proposed for both, SOM and TKM, which evaluates the condition that at an optimum in the weight space the derivative of the quantization error is zero. This quadratic equation can be solved analytically which yields solutions for optimum weights w_i. Apart from sequence recognition tasks, these models have been successfully applied for learning motion-directivity sensitive maps as can be found in the visual cortex [12].

Recursive SOM

The recursive SOM (RecSOM) has been proposed by Voegtlin [50,52] as a mechanism for sequence prediction. The symbols of a given sequence are thereby recursively processed based on the already computed context. Each neuron is equipped with a weight $w_i \in \mathbb{R}^n$ and, additionally, with a context vector $c_i \in \mathbb{R}^N$ which stores an activation profile of the whole map, indicating in which sequential context the vector w_i should arise. Given a sequence $[a_1, \ldots, a_t]$, the activation of neuron n_i at time step j is defined as $d_i(0) = 0$ and

$$d_i(j) = \alpha \cdot \|a_j - w_i\|^2 + \beta \cdot \|(\exp(-d_1(j-1), \ldots, \exp(-d_N(j-1))) - c_i\|^2$$

for $j > 0$, where $\alpha, \beta > 0$ are constants to control mediation between the amount of pattern match versus context match. Hence the respective symbol is compared with the weights a_i. In addition, the already computed context, i.e. the activation of the whole map in the previous time step, has to match with the context c_i of neuron n_i such that the neuron becomes the winner. The comparison of contexts is done by involving the exponential function exp in order to avoid numerical explosion. If this exponential transform was not included, the activation $d_i(j)$ could become huge because the distances with respect to all of the N components of the context could accumulate. The RecSOM has been applied to sequence recognition and prediction of the Mackey-Glass time series and to recognition of sequences induced by a randomized automaton. Training has been done with Hebbian learning in these cases for both, the weights w_i of the neurons and their contexts c_i. Thereby, the parameters (w_i, c_i) of the recursively computed winner neuron n_i are adapted towards the current input a_j and the recursively computed context; the neighbors of n_i are adapted accordingly with a smaller learning rate. The RecSOM shows a good capability of differentiating input sequences. The winners represent different sequences which have been used for training [50,52].

SOM for structured data

The SOM for structured data (SOMSD) has been proposed for the processing of labeled trees with fixed fan-out k [20,46]. The specific case $k = 1$ covers sequences. It is assumed that the neurons are arranged on a rectangular d-dimensional lattice structure. Hence neurons can be enumerated by tuples $\vec{\imath} = (i_1, \ldots, i_d)$ in $\{1, \ldots, N_1\} \times \ldots \times \{1, \ldots, N_d\}$ where $N_i \in \mathbb{N}$ with $N_1 \cdot \ldots \cdot N_d = N$. Each neuron $n_{\vec{\imath}}$ is equipped with a weight $w_{\vec{\imath}}$ and k context vectors $c_1^{\vec{\imath}}, \ldots, c_k^{\vec{\imath}}$ in $\mathbb{R}^d$. They represent the context of a processed tree given by the k subtrees and the indices of their respective winners. The index $I(t)$ of the winning neuron given a tree t with

root label a and subtrees $t_1, \ldots, t_k$ is recursively defined by

$$I(\xi) = (-1, \ldots, -1)$$
$$I(a(t_1, \ldots, t_k)) =$$
$$\text{argmin}_{\vec{\iota}}\{\vec{\iota} \mid \alpha \cdot \|a - w_{\vec{\iota}}\|^2 + \beta \cdot \|I(t_1) - c_1^{\vec{\iota}}\|^2 + \ldots + \beta \cdot \|I(t_k) - c_k^{\vec{\iota}}\|^2\}$$

Hence the respective context of a label in the tree is given by the winners for the k subtrees. Hereby, the empty tree is represented by an index not contained in the lattice, $(-1, \ldots, -1)$. The weights of the neurons consist of compact representations of trees with prototypical labels and prototypical winner indices which represent the subtrees. Starting at the leaves, the winner is recursively computed for an entire tree. Hebbian learning is applied for SOMSD. Starting at the leaves, the index $\vec{\iota}$ of the winner for the subtrees is computed. The attached weights $(w_{\vec{\iota}}, c_1^{\vec{\iota}}, \ldots, c_k^{\vec{\iota}})$ are moved into the direction $(a, I(t_1), \ldots, I(t_k))$ after each recursive processing step, where $I(t_i)$ denotes the winning index of the subtree t_i of the currently processed part of the tree. The neighborhood is updated into the same direction with smaller learning rate.

This method has been used for the classification of artificially constructed pictures represented by tree structured data [19,20]. During learning, a representation of the pictures in the SOMSD emerges which groups together pictures described by similarly structured trees with similar labels. In the reported experiments, the pictures generated by a plex grammar correspond to ships, houses, and policemen with various shapes and colors. SOMSD is capable of properly arranging objects of these categories, e.g. houses. Within the clusters a differentiation with respect to the involved features can be observed like in the standard SOM. Based on this clustering a good classification accuracy of the objects can be obtained by attaching appropriate classes to the nodes [19].

3 A general framework for the dynamic

The main idea for a general framework for these models is derived from the observation that all models share the basic recursive dynamics. They differ in the way how tree structures or sequences, respectively, are internally represented with respect to the recursive processing and the weights of neurons. The above models deal with either sequences over a real-vector space or labeled trees with fan-out k. For convenience, we shortly introduce these data structures formally. Assume L is a set where the labels are taken from. In practice, L is often embedded in some real-vector space. Sequences over L refer to objects $[a_1, \ldots, a_t]$ where $a_i \in L$ and $t \geq 0$ denotes the length of the sequence. If $t = 0$, the empty sequence $[]$ is dealt with. Sequences can be considered as trees with fan-out 1 over L. More generally,

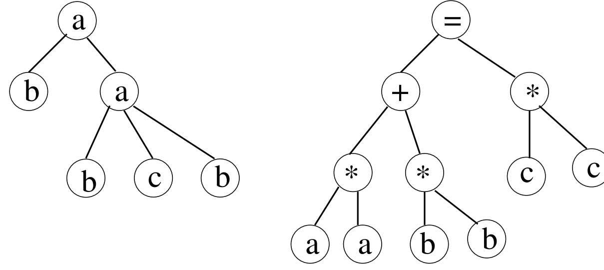

Fig. 1. Example for two trees. Thereby, empty children are omitted. The left tree has fan-out 3, the right one has fan-out 2. Labels are taken from $\{a, b, c, *, =, +\}$. Prefix notation of the left tree is $a(b(\xi,\xi,\xi), a(b(\xi,\xi,\xi), c(\xi,\xi,\xi), b(\xi,\xi,\xi)), \xi)$ resp. $a(b, a(b, c, b))$, if empty trees are omitted in the presentation. The right one is represented by $= (+(*(a, a), *(b, b)), *(c, c))$ (which is $a^2 + b^2 = c^2$) omitting empty trees.

trees with fan-out k over L consist either of the empty tree ξ, or of a root node labeled with an element $a \in L$ and k subtrees $t_1, \ldots, t_k$ over L some of which might be empty. Such a tree is referred to as $t = a(t_1, \ldots, t_k)$. The root node of t is referred to as parent for the root nodes of $t_1, \ldots, t_k$, and the root nodes of $t_1, \ldots, t_k$ are referred to as children of the root node of t. Nodes of the tree which have only empty subtrees are referred to as leaves. A sequence $[s_1, \ldots, s_t]$ corresponds to the tree $s_t(s_{t-1}(\ldots(s_1(\xi)\ldots))$ when the last element of the sequence is taken as root of the tree. Note that terms, logical formulas, and elements of other domains such as chemistry or games can often naturally be represented as tree structures. Two trees representing terms are depicted in Fig. 1.

For simplicity, we will here focus on binary trees with labels in some set W, i.e. each node has at most two children. The generalization to trees with fan-out k, in particular sequences which are trees with fan-out 1 is obvious. Labels of trees may origin from an arbitrary set W, e.g. a real vector space or a discrete set of labels. Let denote by $W^\#$ the set of binary trees with labels in W. We use again a prefix notation if the single parts of a tree are referred to; $a(t_1, t_2)$ represents the tree with root label a and subtrees t_1 and t_2. As above, the empty tree is denoted by ξ. How can this type of data be processed in an unsupervised fashion? We here define the basic ingredients and the basic dynamic or functionality of the map. I.e. given a trained map and a new input tree, how is the winner for this tree computed? Based on this general dynamic, which as a special case includes the models described above, i.e., TKM, RecSOM, and SOMSD, a canonic formulation of Hebbian learning will be derived in the next section.

The processing dynamic is based on the following main ingredients which define the general SOM for structured data or GSOMSD, for short:

(1) The set of labels W together with a similarity measure $d_W : W \times W \to \mathbb{R}$ (usually, images are contained in the non-negative real numbers).
(2) A set of formal representations R for trees together with a similarity measure $d_R : R \times R \to \mathbb{R}$ (again, the image is usually restricted to the non-negative

numbers). Denote the priorly fixed formal representation of the empty tree ξ by r_ξ. The adequate formal representation of any other trees will be computed via the GSOMSD.

(3) A set of neurons N of the self organizing map which we assume to be enumerated with $1, \dots, N = |N|$, for simplicity. A weight function $L = (L_0, L_1, L_2) : N \to W \times R \times R$ attaches a "weight" to each neuron. For each neuron this consists in the map of a triple consisting of a prototype label and two formal representations.

(4) A representation mapping *rep* : $\mathbb{R}^N \to R$ which maps a vector of activations of all neurons into a formal representation.

The idea behind these ingredients is as follows: a tree can be processed iteratively; starting at the leaves, it can be compared with the information contained in GSOMSD until the root is reached. At each step of the comparison the context information resulting from the processing of the two subtrees should be taken into account. Hence a neuron weighted with (w, c_1, c_2) in the map is a proper representation of the entire tree $a(t_1, t_2)$ if the first part of the weight attached to the neuron, w, represents a properly, and c_1 and c_2 represent correct contexts, i.e. they correspond to t_1 and t_2. w and a can be compared using the similarity measure d_W directly. c_1 and c_2 are formal descriptions of the contexts, which could be some pointers or some reduced description for the context and which should be compared with t_1 and t_2, respectively. For this purpose, t_1 and t_2 can be iteratively processed yielding a context, i.e. an activation of all neurons in the map. This activation can be transferred to a formal description of the context via the representation mapping *rep*. The comparison of the output with c_i is then possible using the metric d_R on the formal representations.

This so far verbal description can be formalized by the following definition which allows to compute the *recursive similarity* or activation $\tilde{d}$ of a node n_i given a tree $a(t_1, t_2)$:

$$\tilde{d}(a(t_1, t_2), n_i) = \alpha \cdot d_W(a, L_0(n_i)) + \beta \cdot d_R(R_1, L_1(n_i)) + \beta \cdot d_R(R_2, L_2(n_i))$$

where

$$R_j = \begin{cases} r_\xi & \text{if } t_j = \xi \\ \textit{rep}(\tilde{d}(t_j, n_1), \dots, \tilde{d}(t_j, n_N)) & \text{otherwise} \end{cases}$$

for $j = 1, 2$ and $\alpha, \beta > 0$ are weighting factors. We will refer to this dynamic as *generalized SOM for structured data* or *GSOMSD*, for short. The choice of α and β determines the importance of a proper context in comparison to the correct root label of the representation. The set R enables a precise notation of how trees

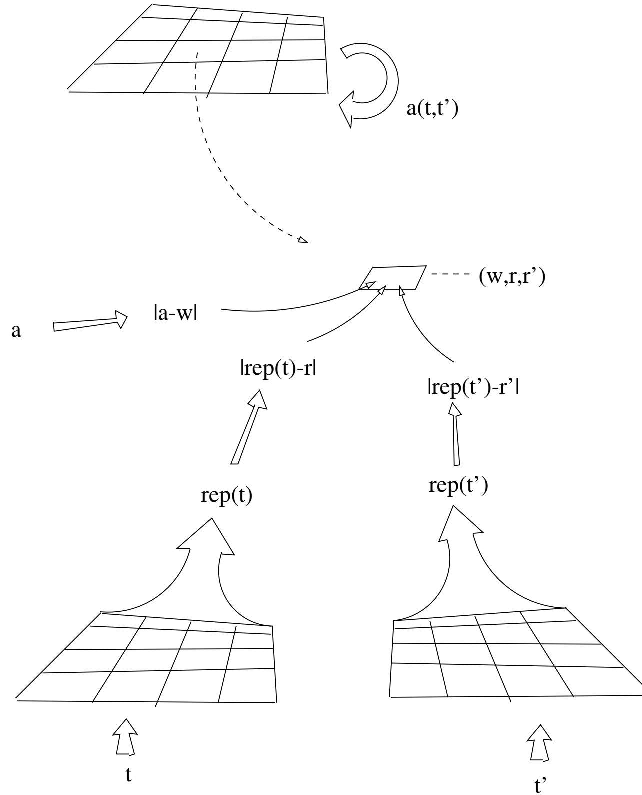

Fig. 2. One recursive processing step: given a tree $a(t, t')$, the distance from a neuron weighted with (w, r, r') is computed weighting the distances of a from w, and the distances of r and r', respectively, from the representations of t and t'. These latter representations can be obtained recursively processing the trees and applying *rep* to the obtained activity profile of the map.

are internally stored in the map. The function *rep* constitutes the interface which maps activity profiles to internal representations of trees. The recursive similarity yields the activation of all neurons for an input tree. Applying *rep*, we can obtain a formal representation of the tree. In order to use the resulting map, for example for information storing and recovering, we can determine a neuron with highest responsibility, i.e. a neuron which fires, or the winner, given the input tree. This could be the neuron with highest or lowest recursive similarity, depending on the meaning of $\tilde{d}$, e.g. depending on the fact whether dot products or distances are used for the computation of the similarities. A picture which explains the processing in one recursive step can be found in Fig. 2.

Note the following:

(1) The labels of the input trees might come from a proper subset of W, e.g. if discrete data are processed, and the representations of the neurons lie in a real vector space. d_W is often the squared standard Euclidian metric or induced by any other metric; however, at this point, we do not need special properties of d_W.

(2) The formal representations R should represent trees in a compact way, e.g. in a finite dimensional vector space. As an example, they could be chosen simply as the index of the neuron with best recursive similarity if a tree is processed with a trained self organizing map. The idea behind is that a connectionistic distributed representation of a tree emerges from the processing of the tree by the map. The formal description could be a shorthand notation, a pointer for this activity profile. Based on this assumption we can compare trees by comparing their formal descriptions.
(3) The neurons can be considered as prototypes for trees, i.e. their root label and their subtrees. The latter are represented by their formal representations. Note that we did not introduce any lattice or topological structure of the neurons at this point. The definition of a topology of the self organizing map does not affect the recursive similarity of neurons given a tree. However, a topological structure might be used for training. In addition, it might prove beneficial for specific applications: a two-dimensional neighborhood structure, for example, allows appropriate visualization of a given map.
(4) The mapping *rep* maps the recursively processed trees which yields an activity profile of the neurons into a formal description of trees. It might be simply the identity or an appropriate compactification, for example. However, it is no longer a symbolic tree representation but a connectionist description based on the activation of the GSOMSD.

Obviously, the GSOMSD can be easily defined for trees with fan-out k where $k \neq 2$. In particular, the case of sequences, i.e. $k = 1$ is therein included. When considering trees with fan-out k, the weight function is of the form $L = (L_0, L_1, \dots, L_k) : N \to W \times R^k$, i.e. k contexts are attached to the neurons corresponding to the fan-out k. The recursive processing reads as

$$\begin{aligned}&\tilde{d}(a(t_1, \dots, t_k), n_i) = \\ &\alpha \cdot d_W(a, L_0(n_i)) + \beta \cdot d_R(R_1, L_1(n_i)) + \dots + \beta \cdot d_R(R_2, L_k(n_i))\end{aligned}$$

where

$$R_j = \begin{cases} r_\xi & \text{if } t_j = \xi \\ rep(\tilde{d}(t_j, n_1), \dots, \tilde{d}(t_j, n_N)) & \text{otherwise} \end{cases}$$

This abstract definition captures the above approaches of structure processing self organizing networks. We proof this fact for the case of binary input trees. The transfer to sequences is obvious.

Theorem 1 *GSOMSD includes the dynamic of SOM, TKM, RecSOM, and SOMSD via appropriate choices of R.*

PROOF. *SOM:* For SOM, we choose $R = \emptyset$ and $d_R \equiv 0$. Then no context is available and hence only the labels of the root of the trees, or elements in W, respectively, are taken into account.

TKM: For the TKM, the ingredients are as follows:

(1) Define $W = \mathbb{R}^n$. d_W is the squared Euclidian metric.
(2) $R = \mathbb{R}^N$ explicitely stores the activation of all neurons if a tree is processed, N denotes the number of neurons. The similarity measure d_R is here given by the dot product. The representation for the empty tree ξ is the vector $r_\xi = (0, \ldots, 0)$.
(3) The weights of the neurons have a special form: in neuron n_i with $L(n_i) = (w, c_1, c_2)$ the first parameter may be an arbitrary value in $\mathbb{R}^n$ obtained by training. The contexts $c_1 = c_2$ coincide with the unit vector which is one at position i and 0 otherwise. Hence the context represented in the neurons is of a particularly simple structure. One can think of the context as a focus: the neuron only looks at its own activation when processing a tree; it doesn't care about the global activation produced by the tree. Mathematically, this is implemented by using the dot product d_R of the context with the unit vector stored by the neuron.
(4) *rep* is simply the identity, no further reduction takes place.

Then the recursive dynamic reduces to the computation $\tilde{d}(a(t_1, t_2), n_i)) = \alpha \cdot \|a - L_0(n_i)\|^2 + \beta \cdot \tilde{d}(t_1, n_i) + \beta \cdot \tilde{d}(t_2, n_i)$ for $t_1, t_2 \neq \xi$. If we choose $\beta = 1 - \alpha$ and consider the case of sequences, this is just a leaky integrator as defined above. An analogous dynamic results for tree structures which does not involve global context processing but focuses on the activation of a single neuron. The winner can be computed as the neuron with smallest value $\tilde{d}$.

RecSOM: For RecSOM, we define:

(1) Define $W = \mathbb{R}^n$. d_W is the squared Euclidian metric.
(2) Define $R = \mathbb{R}^N$ where N is the number of neurons in the self organizing map. d_R is the squared Euclidian metric. Here the formal description of a tree is merely identical to the activation of the neurons in the self organizing map. No reduction with respect to the dimensionality takes place. The representation for the empty tree ξ is the origin $(0, \ldots, 0)$.
(3) The neurons are organized on a lattice in this approach; however, this arrangement does not affect the processing dynamics.
(4) Choose *rep* $=$ *rep*$_R$ where *rep*$_R(x_1, \ldots, x_N) = (\exp(-x_1), \ldots, \exp(-x_N))$. The exponential function is introduced for stability reasons. On the one hand, it prevents the similarities from blowing up during recursive processing; on the other hand, it scales the similarities in a nonlinear fashion to make small similarities getting close to zero. Noise which could disturb the computation due to the large dimensionality of R is suppressed in this way. However, no

information is gained or lost by the application of the exponential function.

These definitions lead to the recursive formula $\tilde{d}(a(t_1,t_2),n_i) = \alpha \cdot \|a - L_0(n_i)\|^2 + \beta \cdot \|R_1 - L_1(n_i)\|^2 + \beta \cdot \|R_2 - L_2(n_i)\|^2$ where $R_j = (\exp(-\tilde{d}(t_j,n_1)), \ldots, \exp(-\tilde{d}(t_j,n_N)))$ for $j = 1,2$ and $t_1, t_2 \neq \xi$. If we restrict the fan-out to 1, we get the dynamic of RecSOM as introduced above. The winner is again computed as the neuron with smallest value $\tilde{d}$.

SOMSD: We choose for SOMSD:

(1) Define $W = \mathbb{R}^n$, d_W is the squared Euclidian metric.
(2) R is the real vector space which contains the set of indices of neurons in the self organizing map. If the neurons lie on a d-dimensional lattice, $R = \mathbb{R}^d$. d_R is the squared Euclidian distance of the lattice points. In addition, the representation of the empty tree ξ is a singular point r_ξ outside the lattice, e.g. $(-1,\ldots,-1)$.
(3) The neurons are connected to a topological structure: the d-dimensional lattice. The weighting attaches appropriate values in $W \times R \times R$ to all neurons. The weights are trained by Hebbian learning. Note that the above assumed enumeration $1,\ldots,N$ of neurons can be substituted by an enumeration based on this lattice with elements $\vec{\iota} \in \{1,\ldots,N_1\} \times \ldots \times \{1,\ldots,N_d\}$.
(4) We choose $rep = rep_S$ as follows: the input domain of rep_S, the set $\mathbb{R}^N$ where N denotes the number of neurons, can be identified by $\mathbb{R}^{N_1 \times \ldots \times N_d}$. rep_S maps a vector of similarities to the index of the neuron with best similarity: $rep_S(x_{(1,\ldots,1)},\ldots,x_{(N_1,\ldots,N_d)}) = \vec{\iota}$ such that $x_{\vec{\iota}}$ is the smallest input. Note that this need not be unique. We assume here that an ordering of the neurons is given. In the case of multiple optima, we assume implicitly that the first optimum is chosen.

With this definition we obtain the general formula $\tilde{d}(a(t_1,t_2),n_i) = \alpha \cdot \|a - L_0(n_i)\|^2 + \beta \cdot \|R_1 - L_1(n_i)\|^2 + \beta \cdot \|R_2 - L_2(n_i)\|^2$ for the recursive similarity for nonempty trees t_1 and t_2 where R_1 and R_2 are the indices of the neurons which are the winners for t_1 and t_2, respectively. The winner of the map can again be computed as the neuron with smallest $\tilde{d}$. □

Note that in all of the above examples $\tilde{d}$ corresponds to a distance which has been computed within the GSOMSD. This is the standard setting which can also be found in the standard SOM: given an input, the neurons compute an activation which measures their distance from the respective input. Afterwards, the winner can be determined as the neuron with smallest distance. Note that this interpretation as distance is common, but more generally we could interprete the similarity $\tilde{d}$ as the activation of the neurons for an input signal. Then, general supervised recurrent and recursive networks are included in the above framework, too, as can be seen by the following argument. Various different dynamics for discrete time

recurrent neural networks (RNN) have been established in the literature [39]. Most of the models can at least be simulated or approximated within the following simple Elman-dynamic which is proved in the reference [21]. Assume that sequences with entries in $\mathbb{R}^n$ are dealt with. The N neurons $n_1, \ldots, n_N$ are equipped with weights $w^i = (w_0^i, w_1^i) \in \mathbb{R}^{n+N}$. Denote by $\tanh$ the hyperbolic tangent. Then the activation of the neurons $\tilde{D} \in \mathbb{R}^N$ given a sequence $[a_1, \ldots, a_t]$ can be computed by

$$\begin{aligned}
&\tilde{D}([\,]) = (0, \ldots, 0), \\
&\tilde{D}([a_1, \ldots, a_t]) = (\tanh(w_0^1 \cdot a_t + w_1^1 \cdot \tilde{D}([a_1, \ldots, a_{t-1}])), \\
&\qquad\qquad \ldots, \tanh(w_0^N \cdot a_t + w_1^N \cdot \tilde{D}([a_1, \ldots, a_{t-1}]))
\end{aligned}$$

where '$\cdot$' denotes the dot product of vectors. Often, a linear function is added in order to obtain the desired output from $\tilde{D}$.

RNNs can be generalized to so-called recursive neural networks (RecNN) to deal with tree structures. Here, we introduce the dynamic for binary trees with labels in $\mathbb{R}^n$. Neurons are weighted with $w = (w_0^i, w_1^i, w_2^i) \in \mathbb{R}^{n+N+N}$. The activation $\tilde{D} \in \mathbb{R}^N$ can recursively be computed by

$$\begin{aligned}
&\tilde{D}(\xi) = (0, \ldots, 0), \\
&\tilde{D}(a(t_1, t_2)) = (\tanh(w_0^1 \cdot a + w_1^1 \cdot \tilde{D}(t_1) + w_2^1 \cdot \tilde{D}(t_2)), \\
&\qquad \ldots, \tanh(w_0^N \cdot a_t + w_1^N \cdot \tilde{D}(t_1) + w_2^N \cdot \tilde{D}(t_2))))
\end{aligned}$$

where again often a linear function is added to obtain the final output.

Theorem 2 *RecNNs can be formulated within the dynamic of GSOMSD.*

PROOF. Choose $\alpha = \beta = 1$ and choose in the recursive computation the following ingredients:

(1) $W = \mathbb{R}^n$, d_W is the dot product.
(2) $R = \mathbb{R}^N$ stores the input net of all neurons, d_R is the dot product, $r_\xi = (0, \ldots, 0)$.
(3) the N neurons are equipped with the weights $L_0(n^i) = w_0^i$, $L_1(n^i) = w_1^i$, and $L_2(n^i) = w_2^i$.
(4) *rep*$(x_1, \ldots, x_N) = (\tanh(x_1), \ldots, \tanh(x_N))$.

In this way we obtain the so-called input net of the neurons as recursive similarity $\tilde{d}$, i.e. $\tanh(\tilde{d}) = \tilde{D}$ where $\tanh$ denotes component-wise application of the hyperbolic tangent. Hence, if we substitute the computation of the final winner

by the function $\tanh$ with a possibly added further linear function, we obtain the processing dynamic of recurrent and recursive networks within the GSOMSD dynamic. $\square$

We could even allow activations of the neurons to be more general than simple one-dimensional vectors and also not corresponding to distances. A dimensionality l of the activation vector of the neurons could be chosen, and $\tilde{d} \in \mathbb{R}^l$ and d_R and d_W would be chosen as l-dimensional vectors. In this case, the sum $\alpha \cdot d_W + \beta \cdot d_R + \beta \cdot d_R$ denotes scalar multiplication and addition of vectors. This setting allows to model RSOM, for example, which, unlike the TKM, stores the leaky integration of the input directions instead of the integration of distances. We obtain an analog of RSOM for binary trees if we choose $R = (\mathbb{R}^N)^l$, $L_1(n_i) = L_2(n_i) = (e_i, \ldots, e_i)$, e_i denoting the i th unit vector in $\mathbb{R}^N$, $d_R((r_1^1, \ldots, r_l^1), (r_1^2, \ldots, r_l^2)) = (r_1^1 \cdot r_1^2, \ldots, r_l^1 \cdot r_l^2)$, *rep* as the identity, and $d_W(a_1, a_2) = a_1 - a_2$. Then the recursive dynamic becomes $\tilde{d}(a(t_1, t_2), n_i) = \alpha(a - L_0(n_i)) + \beta \tilde{d}(t_1, n_i) + \beta \tilde{d}(t_2, n_i)$. For fan-out 1 and $\beta = (1 - \alpha)$, this is the recursive formula of RSOM.

Of course, a computation of the combination of similarities more general than a simple weighted sum is also possible. In this case the computation $\alpha d_W + \beta d_R + \beta d_R : W^2 \times R^2 \times R^2 \to \mathbb{R}^l$ would be substituted by a mapping $\Phi : W^2 \times R^2 \times R^2 \to \mathbb{R}^l$ which maps the inputs given by the actual processed tree and the weights attached to a neuron to the new activation of this neuron. Such more general mechanism could for example model the so-called extended Kohonen feature map (EKFM) or the self-organizing temporal pattern recognizer (SOTPAR) [11,27]. The EKFM equips the SOM with weighted connections w_{ij} from neuron i to j which are trained with a variant of temporal Hebbian learning. Given an input sequence, a neuron n_i can only become active, if it is the neuron with smallest distance from the actual pattern, and, in addition, the weighted sum of all activations of the previous time step, weighted with the weights w_{ji}, is larger than a certain threshold θ_i. It may happen that no winner is found if the sequence to be processed is not known. This corresponds to the fact that the neuron with smallest distance does not become active due to a too small support from the previous time step. This computation can be modeled within the proposed dynamics, given an appropriate choice of Φ, if $\tilde{d}$ is two-dimensional for each neuron, the first part storing the current distances, the second part storing the fact whether the neuron finally becomes active.

SOTPAR activates at each step the neuron with minimum distance from the currently processed label, for which all neurons lying in the neighborhood of a recently activated neuron have a benefit, i.e. they tend to become winner more easily. This is modeled by storing a bias within each neuron, which is initialized with 0 and which is adapted if a neighbored neuron or the neuron itself becomes a winner, and which gradually decreases to 0 over time. This bias is subtracted from the current distance at each time step, such that the effective winner is the neuron with smallest distance and largest bias. Again, this dynamic can easily be modeled within the above

framework if the states of the neurons are two-dimensional, storing the respective distance and the actual bias of each neuron, and if again Φ is chosen appropriately.

In the following, we will focus on the dynamics of GSOMSD, i.e. the states of the neurons are one-dimensional and correspond to distances or similarity values, and Φ has the specific form as defined above. We will discuss Hebbian learning and learning based on an energy function for GSOMSD. Moreover, we will describe the concrete setting for SOMSD and RecSOM adapted for binary trees. These two methods both rely on reasonable contexts, the winner, or the exponentially transformed activation of the entire map, respectively. SOMSD and RecSOM can be seen as two prototypical mechanisms with a different degree of information compression: SOMSD only stores the winner whereas RecSOM stores an activity profile which, due to the exponential transform, focuses on the winners, too. We will not describe the concrete setting for TKM, because this mechanism is restricted with respect to its capability of tree recognition: it can be proved that local contexts as used in TKM are not sufficient for the task of storing trees of arbitrary depth [22].

4 Hebbian learning

In order to find a good neural map, we assume that a finite set of training data $T = \{T_1, \dots, T_d\} \subset W^\#$ is given. We would like to find a neural map such that the training data is represented as accurately as possible by the neural map. In the following, we discuss training methods which adapt the weights of the neurons, i.e. the triples (x, r_1, r_2) attached to the neurons. This means that we assume that only the function L can be changed during training, for simplicity. Naturally, other parts could be adaptive as well, such as the weighting terms α and β, the computation of formal representations *rep*, or even the similarities d_W and d_R. Note that some of the learning algorithms discussed in the following can be immediately transferred to the adaptation of additional parameters involved in e.g. *rep*.

There are various ways of learning in self organizing maps. One of the simplest and most intuitive learning paradigms is Hebbian learning. It has got the advantage of producing simple update formulas and it does not require any further properties of e.g. the functions involved in $\tilde{d}$. Alternatives can be found if an objective of the learning process is explicitely formulated in terms of a cost function which must be optimized. Then gradient descent methods or other optimization techniques, such as a genetic algorithm, can be applied. A gradient descent requires that the involved functions are differentiable and that W and R are real-vector spaces. Actually, most unsupervised Hebbian learning algorithms for simple vectors, although designed as a heuristic method, can be interpreted as a stochastic gradient descent on an appropriate cost function. Hence the two paradigms often yield to the same learning rules for simple vectors. We investigate the situation for structured data in the following.

Here we will first discuss Hebbian learning for structured data and adapt several cost functions for self organizing maps to our approach, afterwards.

We would like to find a proper representation of all trees T_i of the training set T and all its subtrees in the neural map. Since the computation of the recursive similarities for a tree T_i uses the representation of all its subtrees, it can be expected that an adequate representation of T_i can only be found in the neural map if all subtrees of T_i are properly processed, too. Therefore, we assume that for each tree contained in T all its subtrees are contained in T, too. This property of T is essential for efficient Hebbian learning. It can be expected that this property is beneficial for learning based on a cost function as well. Therefore we implicitly assume in the following that the training set T is enlarged to contain all subtrees of each $T_i \in T$. Hebbian learning characterizes the idea of iteratively making the weights of the neuron which fires in response to a specific input more similar to the input. For this purpose some notation of 'moving into the direction of' is required. We assume the following two functions to be given:

- A function $mv_W : W \times W \times \mathbb{R} \to W, (w_1, w_2, \eta) \mapsto mv_W(w_1, w_2, \eta)$ which allows an adaptation of the weights w_1 stored in the first position of a neuron into the direction of the input w_2 by degree η. If W is a real vector space, this function is usually just given by the addition of vectors: $mv_W(w_1, w_2, \eta) = w_1 + \eta(w_2 - w_1)$. If discrete weights not embedded in a real vector space are dealt with, adaptation can be based on a discrete decision:

$$mv_W(a, b, \eta) = \begin{cases} a & \text{if } \eta \leq 0.5 \\ b & \text{otherwise} \end{cases}$$

  If a and b are compared using the edit distance, for example, a specific number of operations which transform a into b could be applied depending on the value of η, as proposed in [18] for the standard SOM.
- A function $mv_R : R \times R \times \mathbb{R} \to R, (r_1, r_2, \eta) \mapsto mv_R(r_1, r_2, \eta)$ which allows us to adapt the formal representations in the same way. If the formal representations are contained in a real vector space as in SOMSD or the recursive SOM, for example, this is given by the addition of vectors, too. Alternatively, we can use discrete transformations if R consists of a discrete set.

Note that we can also handle discrete formal representations in this way. We first formulate Hebbian training for the analogy of SOM-training for the GSOMSD. Here, an additional ingredient, the neighborhood structure or topology is used: assume there is given a neighborhood function

$$nh : N \times N \to \mathbb{R}$$

which measures the distance of two neurons with respect to the topology. Often, a lattice structure is fixed a priori and this lattice of neurons is spread over the

data during training. It might consist of a two-dimensional lattice and the neurons enumerated by $(1,1)$, $(1,2)$, $\ldots$, (N_1,N_2) accordingly. For this two-dimensional lattice $nh(n_{(i_1,j_1)},n_{(i_2,j_2)})$ could be the distance of the indices $|i_1-i_2|+|j_1-j_2|$, for example. However, the lattice structure could be more complex such as a hexagonal grid structure or a grid with exponentially increasing number of neighbors [41]. In this case, an algorithm can be written as follows:

initialize the weights at random

repeat:

choose some training pattern T

for all subtrees $t=a(t_1,t_2)$ *in* T *in inverse topological order:*

compute $\tilde{d}(a(t_1,t_2),n_i)$ *for all neurons* n_i

compute the neuron n_{i_0} *with greatest similarity*

adapt the weights of all neurons n_i *simultaneously:*

$$L_0(n_i) := mv_W(L_0(n_i), a, \eta(nh(n_i,n_{i_0})))$$
$$L_1(n_i) := mv_R(L_1(n_i), R_1, \eta(nh(n_i,n_{i_0})))$$
$$L_2(n_i) := mv_R(L_2(n_i), R_2, \eta(nh(n_i,n_{i_0})))$$

where

$$(*)\quad R_j = \begin{cases} r_\xi & \text{if } t_j=\xi \\ rep(\tilde{d}(t_j,n_1),\ldots,\tilde{d}(t_j,n_N)) & \text{otherwise} \end{cases}$$

for $j\in\{1,2\}$; $\eta:\mathbb{R}\to\mathbb{R}$ is a monotonically decreasing function. Often, $\eta(x)=\eta_0\exp(-x/\sigma^2)$ where $\eta_0>0$ is a learning rate which is decreased at each step in order to ensure convergence; $\sigma>0$ is a term which determines the size of the neighborhood which is taken into account. Usually, σ is decreased during training, too. The dependence $(*)$ requires the recomputation of the recursive similarity for all subtrees of $a(t_1,t_2)$ after changing the weights. This is, of course, very costly. Therefore, the recursive similarity $\tilde{d}$ is usually computed only once for each subtree of a tree, and the values are uniformly used for the update. This approximation does not change training much for small learning rates. Hebbian learning has been successfully applied for both, SOMSD and RecSOM. See for example the literature for the classification of time series or images [19,20,46,50,52].

We would like to point out that the scaling term $\eta(nh(n_i,n_{i_0}))$ might be substituted by a more general function η which depends not only on the winner but on the whole lattice, i.e. it gets as input the distances $\tilde{d}(a(t_1,t_2),n_i)$ of all neurons n_i. This more general setting covers updates based on the soft-min function or the

ranking of all neurons, for example, such as the k-means algorithm.

Two alternatives to SOM proposed in the neural networks literature are vector quantization (VQ) and neural gas (NG) as introduced in Section 2. We obtain an analogy for VQ in the case of structured data if we choose as neighborhood structure

$$\eta(nh(n_i, n_j)) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

This yields a learning algorithm that can handle both, sequences and trees, because of the respective context; however, no topology preservation takes place. A lattice structure is not accounted for by the VQ algorithm. VQ would constitute an alternative to SOM training for the RecSOM. For SOMSD, the metric d_R of the internal representations R in SOMSD would yield random values for VQ which depend on the initialization of the map and the order of presentation of the examples. As a consequence of this argumentation, adaptation of the representations is necessary to apply VQ in this case. Assume the neurons being enumerated by $1, \ldots, N$. Then the alternative $\mathit{rep}_V(x_1, \ldots, x_N) = (0, \ldots, 0, 1, 0, \ldots, 0)$ as the vector with entry 1 at place i iff $x_i \leq x_j$ for all $j \neq i$ instead of rep_R has the effect of decoupling the indices. Note that we can easily apply a form of Hebbian learning to this representations. A fast computation can look like this: choose the similarity measure d_R as the dot product and restrict $R = \{r \in \mathbb{R}^N \mid \sum r_i = 1\}$. We can store the formal representations attached to a neuron in the form $(r_1, \ldots, r_N)/c$ where c is a global scaling factor for the components for each representation. Given a tree $a(t_1, t_2)$ such that neuron n_0 is the winner for $a(t_1, t_2)$ and the neurons n_1 and n_2 are the winners for t_1 and t_2, respectively, we adapt the weights $L_1(n_0)$ and $L_2(n_0)$ in the following way with Hebbian learning: the component representing n_1 and n_2, respectively, is increased by a fixed constant ϵ, the global scaling factors c for the two representations are increased by ϵ. As a consequence, we obtain implicit normalization of the formal representations.

The neural gas algorithm as explained in Section 2 determines the respective neighborhood from the given training example and its distances to all neurons. Hence the update in the above algorithm given a tree t_j becomes

$$\begin{aligned} L_0(n_i) &:= \mathit{mv}_W(L_0(n_i), a, \eta(\mathit{rk}(j, i))) \\ L_1(n_i) &:= \mathit{mv}_R(L_1(n_i), R_1, \eta(\mathit{rk}(j, i))) \\ L_2(n_i) &:= \mathit{mv}_R(L_2(n_i), R_2, \eta(\mathit{rk}(j, i))) \end{aligned}$$

where $\mathit{rk}(j, i)$ denotes the rank of the neuron n_i ordered according to the current distance from the input, i.e. if the tree t_j is currently processed, $\mathit{rk}(j, i)$ denotes the number of neurons n_k such that $\tilde{d}(t_j, n_i) < \tilde{d}(t_j, n_k)$. $\eta(x)$ can for example possess the form $\eta_0 \exp(-x/\sigma^2)$. This update makes sure that neurons which have similar

activation adapt the respective weights into the same direction. Finally, those neurons are neighbors according to this data driven implicit lattice, which constitute the first and second winner for at least one data point. Naturally, the resulting neighborhood structure is less regular. It need not be in accordance with the lattice, and it proposes an alternative to SOM with data oriented lattice. This method is proposed as an alternative lattice for RecSOM in [51]. Like in the case of vector quantization, the representation R in SOMSD requires adaptation. The distance between indices should be dynamically computed during the training algorithm according to the ranking with respect to the given data point. This behavior can be obtained if we choose R as $\mathbb{R}^N$, d_R as the squared Euclidian metric, and if we choose the representation function rep_N where the ith component of $rep_N(x_1, \ldots, x_n)$ is the number of x_j such that $x_j < x_i$, i.e.

$$rep_N(x_1, \ldots, x_n) = (rk(x_1), \ldots, rk(x_N))\,.$$

Hereby, $rk(x_i)$ denotes the number of x_k with $x_k < x_i$.

5 Learning as cost minimization

For SOM, NG, and VQ efforts have been made to derive the Hebbian learning rules alternatively as stochastic gradient methods with appropriate cost functions for simple vector data. Assume that a given differentiable cost function E can be written in the form $\sum_{a_i} E(a_i)$, a_i denoting a training pattern; a stochastic gradient descent with respect to weights w has the form

initialize the weights at random

repeat: choose a training pattern a_i

$$\textit{update } w := w - \eta \cdot \frac{\partial E(a_i)}{\partial w}$$

where $\eta > 0$ is the learning rate. Note that Hebbian learning is similar to this update. It is desirable to prove that Hebbian learning also obeys a stochastic gradient descent with appropriate choice of η. This holds for simple vectors, for example. As a consequence, a prior mathematic objective can be identified and alternative optimization methods, in particular global optimization methods like simulated annealing can be used as an alternative. Moreover, there exist well-known guarantees for the convergence of a stochastic gradient descent if the learning rate η (which may vary over time) fulfills certain properties, see e.g. [35,36].

However, a simple computation shows that the learning for structured data as introduced above cannot be interpreted as an exact gradient descent method. The update

formula for Hebbian learning for the GSOMSD has the form

$$L_0(n_i) := mv_W(L_0(n_i), a, \eta(nh(j,i)))$$
$$L_1(n_i) := mv_R(L_1(n_i), R_1, \eta(nh(j,i)))$$
$$L_2(n_i) := mv_R(L_2(n_i), R_2, \eta(nh(j,i)))$$

where $nh(j,i)$ depends on the neighborhood which is a priorly given lattice, a data driven ranking, or the characteristic function of the winner neuron in the respective cases. For the squared Euclidian metric, the term $mv_R(L_k(n_i), R_k, \eta(nh(j,i)))$ is chosen as $L_k(n_i) - \eta(nh(j,i)) \cdot (L_k(n_i) - R_k)$ for $k \in \{1,2\}$. For general metric d_R, it might be chosen as $L_k(n_i) - \eta(nh(j,i)) \cdot \partial d_R(L_k(n_i), R_k)/\partial L_k(n_i)$. If this formula is interpreted as a stochastic gradient descent, the theorem of Schwartz must hold, i.e.

$$\frac{\partial\left(\eta(nh(j,i)) \cdot \partial d_R(L_k(n_i), R_k)/\partial L_k(n_i)\right)}{\partial L_l(n_m)} = \frac{\partial\left(\eta(nh(j,m)) \cdot \partial d_R(L_l(n_m), R_l)/\partial L_l(n_m)\right)}{\partial L_k(n_i)}$$

is valid for all weights $L_l(n_m)$. The partial derivatives separate into two summands. The first one depends on the derivative of the respective neighborhood $\eta(nh(j,m))$. Thereby only the borders of receptive fields usually contribute and depending on the respective choice of the neighborhood, an equality of the terms on the left and right side can be established like in the case of simple vector data. The second summand of the above derivative yields the terms $\eta(nh(j,i)) \cdot \partial^2 d_R(L_k(n_i), R_k)/\partial L_k(n_i)\partial L_l(n_m)$ for the left side and for the right side we obtain $\eta(nh(j,m)) \cdot \partial^2 d_R(L_l(n_m), R_l)/\partial L_l(n_m)\partial L_k(n_i)$. Note that R_l depends on $L_k(n_i)$ and R_k depends on $L_l(n_m)$. Hence the two terms do usually not coincide: R_l and R_k are entirely different functions for the weights. For VQ, for example, the term $\eta(nh(j,i))$ is nonvanishing only for $i = j$, in which case the above second derivative yields a nonvanishing contribution, whereas the product vanishes for all choices $i \neq j$. Hence Hebbian learning for structured data does not in general correspond to gradient descent.

We would like to investigate which learning rules result for appropriate cost functions and how they relate to Hebbian learning. We first have a look at SOM, VQ, and NG for simple vectors. Denote the simple labels or vectors used for training by a_i. The weights of the map are denoted by w_j. The cost function of VQ for the case of simple vectors has the form

$$E_V = \frac{1}{2}\sum_i \sum_j \chi(a_i, w_j)\|a_i - w_j\|^2$$

where $\chi(a_i, w_j)$ denotes the characteristic function of the receptive field of w_j, i.e. it is 1 if n_j is the winner for a_i and 0 otherwise. Taking the derivatives with respect to the weights w_j yields the learning rule of VQ. Note that the derivatives of χ are computed using the δ-function. We ignore this point for simple vectors and provide a detailed derivation of the formulas including the borders of χ for the structured case. The cost function of NG is (up to constants) of the form [38]

$$E_N = \frac{1}{2}\sum_i \sum_j \eta(rk(i,j))\|a_i - w_j\|^2$$

where $rk(i,j)$ denotes the rank of neuron n_j if the neurons are sorted according to their distance from the given data point. SOM itself does not possess a cost function which could be transferred to the continuous case, i.e. if a data distribution instead of a finite training set is considered [10]. For the discrete case (i.e. a finite data set), an energy function can be constructed [42,43]. The article [25] proposes a cost function for a slightly modified version of SOM: even for the continuous case

$$E_S = \frac{1}{2}\sum_i \sum_j \chi_S(i,j) \sum_k \eta(nh(n_j, n_k))\|a_i - w_k\|^2$$

where $\chi_S(i,j)$ denotes the characteristic function of the receptive field of the winner, and a slightly modified definition of the winner is used:

$$\chi_S(i,j) = \begin{cases} 1 & \text{if } \sum_k \eta(nh(n_j,n_k)\|a_i - w_k\|^2 \\ & \qquad \leq \sum_k \eta(nh(n_{j'},n_k)\|a_i - w_k\|^2 \text{ for all } j' \\ 0 & \text{otherwise} \end{cases}$$

Hence the winner in the above cost function as well as the modified learning rule of SOM is not the neuron with smallest distance but the neuron with smallest averaged distance with respect to the local neighborhood.

Obviously, all of the above cost functions have the form

$$E = \sum_i f(\|a_i - w_1\|^2, \dots, \|a_i - w_N\|^2)$$

with a function f chosen according to the specific setting, e.g. the function equals $f(\|a_i - w_1\|^2, \dots, \|a_i - w_N\|^2) = \frac{1}{2}\sum_j \chi(a_i, w_j)\|a_i - w_j\|^2$ for VQ. This general scheme allows an immediate transfer of the above cost functions to the structured case: the term $\|a_i - w_j\|^2$ is substituted by the respective recursive similarity of the neurons for a given tree structure. Given a neural map for structured data with

neurons n_i and a training set $T = \{T_1, \dots, T_d\}$ which is complete with respect to subtrees, the general cost function has the form

$$E = \sum_{t_i \in T} f(\tilde{d}(T_i, N))$$

where $\tilde{d}(T_i, N)$ denotes the vector $(\tilde{d}(T_i, n_1), \dots, \tilde{d}(T_i, n_N))$. If f is chosen corresponding to the above cost functions of VQ, NG, or SOM, this measure can be taken as an objective for the structured case. Taking the derivatives we obtain formulas for a stochastic gradient descent in the structured case. It will be shown that Hebbian learning as introduced above constitutes an approximation of a stochastic gradient descent for the cost functions of NG, VQ, or SOM which disregards the contributions in the error function due to substructures.

We start computing the derivatives of the above general function $(*)$ with respect to the weights of the neurons. For this purpose, we assume that W and R are contained in a real vector space, and we assume that all involved functions including f, *rep*, d_R, and d_W are differentiable. (They are not e.g. for SOMSD, we will discuss this point later.) The derivative with respect to a weight $L_I(n_J)$ for $I \in \{0, 1, 2\}$ and $J \in \{1, \dots, N\}$ yields

$$\frac{\partial f(\tilde{d}(T_i, N))}{\partial L_I(n_J)} = \sum_{j=1}^{N} \frac{\partial f(\tilde{d}(T_i, N))}{\partial \tilde{d}(T_i, n_j)} \cdot \frac{\partial \tilde{d}(T_i, n_j)}{\partial L_I(n_J)}$$

The first part depends on the respective cost function, i.e. the choice of f. It often involves the use of δ-functions for computing the derivatives because f is built of characteristic functions. The second component can be computed as follows: Define $\partial_i f(x_1, \dots, x_n) := \partial f(x_1, \dots, x_n)/\partial x_i$ for $i \leq n$ as a shorthand notation. If $f = (f_1, \dots, f_m)$ is a vector of functions, then the term $\partial_i f(x_1, \dots, x_n)$ denotes the vector $(\partial_i f_1(x_1, \dots, x_n), \dots, \partial_i f_m(x_1, \dots, x_n))$. If x_i is a vector $(x_{i1}, \dots, x_{im})$, we define $\partial_i f(x_1, \dots, x_n) = (\partial f(x_1, \dots, x_n)/\partial x_{i1}, \dots, \partial f(x_1, \dots, x_n)/\partial x_{im})$. As above, we use the abbreviation

$$R_j = \begin{cases} r_\xi & \text{if } t_j = \xi \\ \mathit{rep}(\tilde{d}(t_j, n_1), \dots, \tilde{d}(t_j, n_N)) & \text{otherwise} \end{cases}$$

for $j \in \{1, 2\}$ and we denote the corresponding derivative with respect to a variable x by

$$\partial R_{j,x} = \begin{cases} 0 & \text{if } t_j = \xi \\ \sum_{l=1}^{N} \partial_l \mathit{rep}(\tilde{d}(t_j, n_1), \dots, \tilde{d}(t_j, n_N)) \cdot \partial \tilde{d}(t_j, n_l)/\partial x & \text{otherwise} \end{cases}$$

for $j \in \{1, 2\}$. Then we have

$$\begin{aligned}
\partial\tilde{d}(a(t_1,t_2),n_j)/\partial L_0(n_J) &= \alpha\,\delta_{jJ}\,\partial_2 d_W(a, L_0(n_j)) && (1.1)\\
&+ \beta\,\partial_1 d_R(R_1, L_1(n_j)) \cdot \partial R_{1,L_0(n_J)} && (1.2)\\
&+ \beta\,\partial_1 d_R(R_2, L_2(n_j)) \cdot \partial R_{2,L_0(n_J)} && (1.3)
\end{aligned}$$

for the first components of the weights where $\delta_{ij} \in \{0, 1\}$ is the Kronecker symbol with $\delta_{ij} = 1 \iff i = j$. '$\cdot$' denotes the dot product. For the formal representations attached to a neuron we find

$$\begin{aligned}
\partial\tilde{d}(a(t_1,t_2),n_j)/\partial L_1(n_J) &= \beta\,\delta_{jJ}\,\partial_2 d_R(R_1, L_1(n_j)) && (2.1)\\
&+ \beta\,\partial_1 d_R(R_1, L_1(n_j)) \cdot \partial R_{1,L_1(n_J)} && (2.2)\\
&+ \beta\,\partial_1 d_R(R_2, L_2(n_j)) \cdot \partial R_{2,L_1(n_J)} && (2.3)
\end{aligned}$$

and

$$\begin{aligned}
\partial\tilde{d}(a(t_1,t_2),n_j)/\partial L_2(n_J) &= \beta\,\delta_{jJ}\,\partial_2 d_R(R_2, L_2(n_j)) && (3.1)\\
&+ \beta\,\partial_1 d_R(R_1, L_1(n_j)) \cdot \partial R_{1,L_2(n_J)} && (3.2)\\
&+ \beta\,\partial_1 d_R(R_2, L_2(n_j)) \cdot \partial R_{2,L_2(n_J)} && (3.3)
\end{aligned}$$

Hence the derivatives can be computed recursively over the depth of the tree. Starting at the leaves, we obtain formulas for the derivatives of the respective activation with respect to the weights of the neurons. The complexity of this method is of order $NTWR$, N denoting the number of neurons, T the number of labels of the tree, W the dimensionality of the weights of each neuron, and R the dimensionality of R.

Note that the first summands of the derivatives yield terms which occur in Hebbian learning, too: $\partial_2 d_R(a, b)$ and $\partial_1 d_R(a, b)$ coincide for the squared Euclidian metric with the terms $-2(a - b)$ or $2(a - b)$, respectively. Hence we get the original Hebbian learning rules if we only consider the first summands and involve the respective terms arising from f for NG, VQ, and SOM. Hebbian learning disregards the contribution of the subtrees because it drops the latter two summands in the above recursive formulas for the derivatives. Hence unlike the simple vector case where Hebbian learning and a stochastic gradient descent coincide, structured data causes differences for the methods. However, the factor β is usually smaller than 1; i.e. the weighting factor for the two additional terms in the above gradient formulas vanishes exponentially with the depth since it is multiplied in each recursive step by the small factor β. Therefore, Hebbian learning can be seen as an efficient approximation of the precise stochastic gradient descent.

Nevertheless, the formulation as cost minimization allows to also formulate supervised learning tasks within this framework. Assume a desired output y_i is given for each tree T_i. Then we can train a GSOMSD which approximately yields the output y_i given T_i if we minimize $\sum_i (y_i - \tilde{d}(T_i, N))^2$ with a stochastic gradient descent. We have already shown that supervised recursive networks can be simulated within the GSOMSD dynamic. The introduction of the above error function introduces training methods for these cases, too. The respective formulas are in this case essentially equivalent to the well-known real time recurrent learning and its structured counterpart [40]. Here a Hebbian approximation would yield training with so-called truncated gradient. Note that the well-known problem of long-term dependencies, i.e. the difficulty to latch information through several recursive layers, arises in this context [4,26]. This problem of long term dependencies is obviously avoided in Hebbian learning because the map is always trained for all substructures of each structure in the given data set. This allows us to drop the terms of the gradients which include vanishing gradients.

The reference to RTRL proposes to also formulate gradient descent on the above error functions in termini of another standard algorithm of recurrent and recursive network training, backpropagation through time or structure, respectively (BPTT and BTTS) [13,40]. BPTT has a lower complexity with respect to the number of neurons. The idea of BPTT is to use bottlenecks of the computation, i.e. a parameterization of the cost function including few variables which can substitute the contribution of the activation of all neurons to the respective activation of the neurons in later time steps. Then using the chain rule, the error signals can be backpropagated efficiently. We assume that the representation function *rep* is R-dimensional with components $\mathit{rep}_1, \ldots, \mathit{rep}_R$. Consider a tree T. For simplicity, we refer to the nodes in T by their labels a_i and we identify the subtree with root a_i and the root label in the notation. The two subtrees of a node a_i are denoted by $\mathit{left}(a_i)$ and $\mathit{right}(a_i)$, its parent node in T is denoted by $\mathit{parent}(a_i)$. We denote the formal representation of a tree a_i by $R(a_i)$ and its components by $R_j(a_i)$. Each weight $L_i(n_j)$ of the GSOMSD is used multiple times in a recursive computation. We assume a corresponding number of identical copies $L_i(n_j)(a_k)$ is given, where $L_i(n_j)(a_k)$ refers to the copy which is used in the last recursive step for computing $\tilde{d}(a_k, N)$. The part of the cost function which comes from labels in the tree T is denoted by $E(T) = \sum_{a_k \in T} f(\tilde{d}(a_k, N))$. The derivative with respect to $L_i(n_j)$ can be computed as the sum of all derivatives with respect to one copy $L_i(n_j)(a_k)$. A bottleneck for the computation of these derivatives corresponds to the inputs net of BPTT and, in the case of RNNs, yields the respective BPTT formula: we use the similarities given as output of d_R and d_W in the computation as bottleneck. Define

$$D_I(a_j, n_i) = \begin{cases} d_W(a_j, L_0(n_i)), & \text{if } I = 0, \\ d_R(R(\mathit{left}(a_j)), L_1(n_i)), & \text{if } I = 1, \\ d_R(R(\mathit{right}(a_j)), L_2(n_i)), & \text{if } I = 2 \end{cases}$$

We find for one copy $L_i(n_j)(a_k)$:

$$\frac{\partial E(T)}{\partial L_i(n_j)(a_k)} = \frac{\partial E(T)}{\partial dR_i(a_k,n_j)} \begin{cases} \partial_2 d_W(a_k, L_0(n_j)), & i=0, \\ \partial_2 d_R(R(\mathit{left}(a_k)), L_1(n_j)), & i=1, \\ \partial_2 d_R(R(\mathit{right}(a_k)), L_2(n_j)), & i=2, \end{cases}$$

where

$$\frac{\partial \tilde{d}(a_k,n_j)}{\partial L_i(n_j)(a_k)} = \begin{cases} \alpha \cdot \partial_2 d_W(a_k, L_0(n_j)) & \text{if } i=0 \\ \beta \cdot \partial_2 d_R(R(\mathit{left}(a_k)), L_1(n_j)) & \text{if } i=1 \\ \beta \cdot \partial_2 d_R(R(\mathit{right}(a_k)), L_2(n_j)) & \text{if } i=2 \end{cases}$$

The derivative of $E(T)$ with respect to $D_i(a_k, n_j)$ can be computed via backpropagation by

$$\frac{\partial E(T)}{\partial D_0(a_k,n_j)} = \frac{\partial f(\tilde{d}(a_k,N))}{\partial \tilde{d}(a_k,n_j)} \cdot \alpha$$

and for $i \in \{1,2\}$:

$$\frac{\partial E(T)}{\partial D_i(a_k,n_j)} = \frac{\partial f(\tilde{d}(a_k,N))}{\partial \tilde{d}(a_k,n_j)} \cdot \beta + \sum_l \frac{\partial E(T)}{\partial D_I(\mathit{parent}(a_k),n_l)} \beta \partial_1 d_R(R(a_k), L_I(n_l)) \cdot \partial_j \mathit{rep}(\tilde{d}(a_k,N))$$

where $I \in \{1,2\}$ is 1 iff a_k is the left child of $\mathit{parent}(a_k)$. $\partial f(\tilde{d}(a_k,N))/\partial \tilde{d}(a_k,n_j)$ depends on the specific cost function. This backpropagation is of order $T(W + N^2R)$. Note that for a function of the form $\mathit{rep}(x_1,\ldots,x_N) = (\mathit{rep}_1(x_1), \ldots, \mathit{rep}_N(x_N))$ the complexity further reduces because the derivative $\mathit{rep}_i(x_1,\ldots,x_n)$ reduces to a vector with only one nonzero entry. Hence the complexity in this case reduces to order $T(W + N^2)$, the complexity of BPTT.

Another possible bottleneck can be found if the dimensionality of *rep* is low such as in SOMSD. Then backpropagation can be done through the internal representations of the structured data. The complexity of this method depends on the dimensionality of R. Explicit formulas can be found in [22].

We finally look at the specific cost functions for VQ, (approximate) SOM, and NG for the structural case in order to show explicitly that Hebbian learning can be interpreted as approximate stochastic gradient descent in these cases.

Energy function of VQ

The cost function of vector quantization in our case is

$$E_V(T) := \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{N}\chi(T_i,n_j)\tilde{d}(T_i,n_j)$$

where

$$\chi(T_i,n_j) = \begin{cases} 1 & \text{if } \tilde{d}(T_i,n_j) \text{ is minimal} \\ 0 & \text{otherwise} \end{cases}$$

Hence $f(\tilde{d}(T_i,N))$ has the form $\frac{1}{2}\sum_j \chi(T_i,n_j)\tilde{d}(T_i,n_j)$. The only term in the above derivatives which is specific for the cost function of VQ is $\partial f(\tilde{d}(a_k,N))/\partial\tilde{d}(a_k,n_j)$. The derivative of this term can be found in the appendix. If we disregard the terms (1.2), (1.3), (2.2), (2.3), (3.2), and (3.3) in the general formulas for the derivatives, we obtain the Hebb formulas. Note thereby, that these terms contain $\alpha^i\beta$ or α^{i+1} for $i \geq 1$, which is usually small. If d_W and d_R constitute the Euclidian metric, these truncated Hebbian formulas look like in the following:

$$\begin{aligned} L_0(n_l) &:= L_0(n_l) + \eta\cdot\alpha\cdot\xi(T_i,n_l)(a - L_0(n_l)) \\ L_1(n_l) &:= L_1(n_l) + \eta\cdot\beta\cdot\xi(T_i,n_l)(R_1 - L_1(n_l)) \\ L_2(n_l) &:= L_2(n_l) + \eta\cdot\beta\cdot\xi(T_i,n_l)(R_2 - L_1(n_l)) \end{aligned}$$

where a denotes the current label and R_i the internal representations of the subtrees.

It is possible to transfer the above gradient calculations to RecSOM directly because the formal representation for trees lies in a real vector space and the involved functions are differentiable. The component number l of $\partial_l rep_R(x_1,\ldots,x_N)$ yields $-\exp(-x_l)$, all other components are zero. For SOMSD, the above method cannot be applied directly: the involved function rep_S is not differentiable; moreover, it is expected that rep_S is substituted by rep_V as before, which encodes winners in a unary fashion instead of referring to a lattice location. Although rep_V is not differentiable, it can be approximated up to any desired degree of accuracy with a differentiable function, e.g. using the soft-min function

$$rep_V(x_1,\ldots,x_N) \approx \left(\frac{\exp(-x_1/\gamma)}{\sum_l \exp(-x_l/\gamma)},\ldots,\frac{\exp(-x_N/\gamma)}{\sum_l \exp(-x_l/\gamma)}\right)$$

where $\gamma > 0$ controls the quality of the approximation. The original function is recovered for $\gamma \to 0$ except for situations where the minimum is not unique.

Energy function of SOM

As before we adapt a variation proposed in [25] which uses a slightly different notation of the winner. This is done because the original learning rule as proposed by Kohonen does not possess a cost function in the continuous case (though for the discrete case an energy function exists [42]). The corresponding cost function for structured data has the form

$$E_S(T) = \frac{1}{2} \sum_{i=1}^{T} \sum_{j=1}^{N} \chi_S(T_i, n_j) \sum_{k=1}^{N} \eta(nh(n_j, n_k)) \tilde{d}(T_i, n_k)$$

where $nh : N \times N \to \mathbb{R}$ describes the neighborhood function of the lattice and $\eta(x) = \eta_0 \exp(-x/\sigma^2)$ with $\eta_0 > 0$, $\sigma > 0$ provides a scaling factor for the update of neighbors.

$$\chi_S(T_i, n_j) = \begin{cases} 1 & \text{if } \sum_{k=1}^{N} \eta(nh(n_j, n_k)) \tilde{d}(T_i, n_k)^2 \text{ is minimal} \\ 0 & \text{otherwise} \end{cases}$$

determines the winner. Here the winner is the neuron with optimum weighted distance with respect to the neighborhood. Given a tree T_i, this cost function induces the function

$$f(\tilde{d}(T_i, N)) = \frac{1}{2} \sum_{j} \chi_S(T_i, n_j) \sum_{k} \eta(nh(n_j, n_k)) \tilde{d}(T_i, n_k)$$

The derivative with respect to $\tilde{d}(T_i, n_l)$ can be found in the appendix. Disregarding the terms (1.2), (1.3), (2.2), (2.3), (3.2), and (3.3) this yields the standard Hebb terms if the Euclidian metric d_R and d_W is used:

$$L_0(n_l) := L_0(n_l) + \eta \cdot \alpha \cdot \textstyle\sum_k \xi(T_i, n_k) \eta(nh(n_k, n_l))(a - L_0(n_l))$$
$$L_1(n_l) := L_1(n_l) + \eta \cdot \beta \cdot \textstyle\sum_k \xi(T_i, n_k) \eta(nh(n_k, n_l))(R_1 - L_1(n_l))$$
$$L_2(n_l) := L_2(n_l) + \eta \cdot \beta \cdot \textstyle\sum_k \xi(T_i, n_k) \eta(nh(n_k, n_l))(R_2 - L_1(n_l))$$

The specific adaptations of this method for the recursive SOM and SOMSD follow the same line as in the previous case. Note that we could again use the alternatives of forward or backward propagation as described above. Again the function rep_S

of SOMSD is not differentiable. However, we can approximate the function *rep* up to every desired degree. Denote the indices of the neurons by $\vec{\iota}(1), \ldots, \vec{\iota}(N)$. The representation function which computes the index of the minimum input can be approximated by the soft-min function

$$\mathit{rep}_S(x_1, \ldots, x_n) \approx \sum_{i=1}^{N} \vec{\iota}(i) \cdot \frac{\exp(-x_i/\gamma)}{\sum_l \exp(-x_l/\gamma)}$$

where $\gamma > 0$ controls the quality of the approximation. This approximation is differentiable. The original function is recovered for $\gamma \to 0$ except for situations where the minimum is not unique.

Note that for SOMSD the lattice of neurons has to be chosen a priori. It can be expected that a two-dimensional regular grid is appropriate only for very restricted data sets due to the incompatibility between the grid topology and the complex data. The experiments of [20] show that different structures are stored in different clusters in a regular grid map. The lattice can hold different structures only partially due to a combinatorial explosion of possible neighbors: an exponential increase can be observed if a small tree is expanded to trees of more complex structures. One could expect that lattices which reflect this property of an exponentially increasing neighborhood, like the hyperbolic SOM [41], could be a good alternative choice.

Energy function of NG

The cost function of NG adapted to structured data has the following form:

$$E_N(T) = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{N} \eta(\mathit{rk}(i,j)) \tilde{d}(T_i, n_j) \,.$$

$\mathit{rk}(i,j)$ equals the number of neurons k such that $\tilde{d}(T_i, n_k) < \tilde{d}(T_i, n_j)$. Again $\eta(x) = \eta_0 \exp(-x/\sigma^2)$ is a scaling factor. If we put this cost function in the above general form, we find for the function f:

$$f(\tilde{d}(T_i, N)) = \frac{1}{2} \sum_j \eta(\mathit{rk}(i,j)) \tilde{d}(T_i, n_j) \,.$$

The derivative with respect to $\tilde{d}(T_i, n_l)$ is again given in the appendix. The Hebbian updates result if (1.2), (1.3), (2.2), (2.3), (3.2), and (3.3) are discarded and the

Euclidian metric d_R and d_W is used:

$$\begin{aligned}
L_0(n_l) &:= L_0(n_l) + \eta \cdot \alpha \cdot \eta(rk(i,l))(a - L_0(n_l)) \\
L_1(n_l) &:= L_1(n_l) + \eta \cdot \beta \cdot \eta(rk(i,l))(R_1 - L_1(n_l)) \\
L_2(n_l) &:= L_2(n_l) + \eta \cdot \beta \cdot \eta(rk(i,l))(R_2 - L_1(n_l))
\end{aligned}$$

The distance between indices should be dynamically computed during the training algorithm for SOMSD according to the ranking with respect to the respective data point. We could use the representation function $rep_N(x_1, \ldots, x_n) =$

$$(rk(x_1), \ldots, rk(x_N)) = \left(\sum_{j=1}^{N} \mathrm{H}(x_1 - x_j), \ldots, \sum_{j=1}^{N} \mathrm{H}(x_N - x_j) \right)$$

where $rk(x_i)$ denotes the rank of x_i if the values $x_1, \ldots, x_N$ are ordered according to their size. Since this function is not differentiable, we have to approximate by a differentiable function, e.g. we can substitute H by a sigmoidal approximation $\mathrm{sgd}_\gamma(x) := \mathrm{sgd}(x/\gamma) \to \mathrm{H}(x)$ for $\gamma > 0$, $\gamma \to 0$ and $x \neq 0$ where $\mathrm{sgd}(x) = (1 + \exp(-x))^{-1}$. Note that the same amount is laid on each position of the ranking in this formulation. It can be expected that the position of winning units is more important than the rank of neurons which are far away. Hence one could modify the formal representation of trees such that the winning neurons are ranked higher. One possibility is to include an exponential weighting $rep(x_1, \ldots, x_n) = (\exp(-rk_k(x_1)), \ldots, \exp(-rk_k(x_N)))$. This has the effect that deviations for neurons which are close to the currently presented tree are ranked higher than neurons which are further away.

6 Discussion

We have proposed a general framework for unsupervised processing of structured data based on the main idea of recursive processing of the given recursive structured data. For this purpose, the dynamics of supervised recurrent and recursive networks is directly transferred to the unsupervised framework. Many special approaches like TKM, RecSOM, SOMSD, and even recurrent and recursive networks are covered by the framework. A key issue of the dynamics is the notion of internal representations of context, which enables networks to store activation profiles of recursively processed substructures in a distributed manner in a finite and fixed dimensional vector space. This allows the comparison of structured data of arbitrary size. The general framework allows to formulate training mechanisms in a uniform manner. Hebbian learning with various topologies such as VQ and NG topology can be immediately formulated. It turns out that unlike the case of unsupervised

vector processing, Hebbian learning is only an approximation of a gradient dynamics of appropriate cost functions. We have formulated the cost functions for the general framework and we derived two ways to precisely compute the gradients. Hebbian learning disregards the contribution of substructures in all cases and is thus much less costly than the precise approaches. Nevertheless, the precise formulation allows us to recover supervised training mechanisms within the general framework, too. An appropriate choice of the cost functions yields standard training algorithms for supervised recurrent networks like BPTT and RTRL. Moreover, this formulation proposes how to transfer different learning paradigms such as generalized vector quantization and variations to the recursive case [23,44]: learning vector quantization (LVQ) [31,32] constitutes a self-organizing supervised training method to learn a prototype based clustering of data with Hebb-style learning rules. The approach [44] proposes a cost function for variants of LVQ and introduces so-called GLVQ. This method constitutes a very stable and intuitive learning method for which additional features like automatic metric adaptation have been developed [23]. Since all these methods rely on cost functions which only depend on the squared Euclidian distance of patterns from the weights of neurons, they can immediately be included in the above framework for structured data, and above formulas can be used for calculating the corresponding derivatives.

Starting from this general formulation, a uniform investigation of properties of recurrent self organizing maps is possible. We already demonstrated the possibility with respect to the investigation of training algorithms. Further directions of research can take general properties of *rep* and the internal representation of trees into consideration which would help to design general criteria of the respective functions and uniform possibilities to evaluate the approaches. First steps into this direction can be found in [22] where the noise tolerance of various representations, the representation capability, and the notion of topology preservation are discussed with specific emphasis on the respective internal representation. Note that standard issues of self-organizing maps include the problem of convergence and ordering, the notion of topology preservation, and the magnification as discussed e.g. in [6,42,24]. Analogous issues can be uniformly investigated for general recursive self-organizing maps.

References

[1] A.M. Bianucci, A. Micheli, A. Sperduti, and A. Starita. Application of cascade correlation networks for structures to chemistry. *Journ. of Appl. Int.*, 12:117–146, 2000.

[2] P. Baldi, S. Brunak, P. Frasconi, G. Pollastri, and G. Soda. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11), 1999.

[3] G. Barreto and A. Araújo. Time in self-organizing maps: An overview of models. *Int.*

Journ. of Computer Research, 10(2):139–179, 2001.

[4] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE TNN*, 5(2):157–166, 1994.

[5] G. Chappell and J. Taylor. The temporal Kohonen map. *Neural Networks* 6:441-445, 1993.

[6] M. Cottrell, J.C. Fort, and G. Pagèt. Theoretical aspects of the SOM algorithm. *Neurocomputing* 21:119–138, 1998.

[7] M. Cottrell and P. Letrémy. Analyzing surveys using the Kohonen algorithm. In M. Verleysen (ed.): European Symposium on Artificial Neural Networks 2003, pages 85–92, d-side publication, 2003.

[8] M. Cottrell and P. Rousset. The Kohohnen algorithm: a powerful tool for analysing and representing multidimensional quantitative and qualitative variables. In: Proceedings IWANN'97, pages 861–871, Springer, 1997.

[9] F. Costa, P. Frasconi, V. Lombardo, and G. Soda. Towards incremental parsing of natural language using recursive neural networks. To appear in: *Applied Intelligence*.

[10] E. Erwin, K. Obermayer, and K. Schulten. Self-organizing maps, convergence properties, and energy functions. *Biological Cybernetics* 67(1):47–55, 1992.

[11] N.R. Euliano and J.C. Principe. A spatiotemporal memory based on SOMs with activity diffusion. In E.Oja and S.Kaski (eds.), *Kohonen Maps*, Elsevier, 1999.

[12] I. Farkas and R. Miikkulainen. Modeling the self-organization of directional selectivity in the primary visual cortex. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 251-256, Springer, 1999.

[13] P. Frasconi, M. Gori, A. Küchler, and A. Sperduti, A field guide to dynamical recurrent networks, in: J.F.Kolen, S.C.Kremer(eds.), *From Sequences to Data Structures: Theory and Applications*, pages 351-374, IEEE, 2001.

[14] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE TNN*, 9(5):768–786, 1998.

[15] P. Frasconi, M. Gori, and A. Sperduti. Learning Efficiently with Neural Networks: A Theoretical Comparison between Structured and Flat Representations. In: *ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence*, IOS Press, 2000.

[16] C. L. Giles, G. M. Kuhn, and R. J. Williams. Special issue on dynamic recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(2), 1994.

[17] M. Gori, M. Mozer, A. C. Tsoi, and R. L. Watrous. Special issue on recurrent neural networks for sequence processing. *Neurocomputing*, 15(3-4), 1997.

[18] S. Günter and H. Buhnke. Validation indices for graph clustering. In: J.-M. Jolion, W.G. Kropatsch, and M. Vento (eds.), *Proceedings of the third IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 229-238, Ischia, Italy, 2001.

[19] M. Hagenbuchner, A.C. Tsoi, and A. Sperduti. A supervised self-organizing map for structured data. In N.Allison, H.Yin, L.Allinson, and J.Slack (eds.), *Advances in Self-Organizing Maps*, pages 21-28, Springer, 2001.

[20] M. Hagenbuchner, A. Sperduti, and A.C. Tsoi. A Self-Organizing Map for Adaptive Processing of Structured Data. *IEEE Transactions on Neural Networks*, 14(3):491–505, 2003.

[21] B. Hammer. *Learning with Recurrent Neural Networks*. LNCIS 254, Springer, 2000.

[22] B. Hammer, A. Micheli, and A. Sperduti. *A general framework for self-organizing structure processing neural networks*. Technical report TR-03-04, Dipartimento di Informatica, Università di Pisa, 2003.

[23] B. Hammer and T. Villmann. Generalized relevance learning vector quantization. *Neural Networks* 15(8-9):1059-1068, 2002.

[24] B. Hammer and T. Villmann. Mathematical aspects of neural networks. In M. Verleysen (ed.): European Symposium of Artificial Neural Networks 2003, pages 59–72, 2003.

[25] T. Heskes. Self-organizing maps, vector quantization, and mixture modeling. *IEEE Transactions on Neural Networks*, 12:1299-1305, 2001.

[26] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neur. Comp.*, 9(8):1735–1780, 1997.

[27] A. Hoekstra and M.F.J. Drossaers. An extended Kohonen feature map for sentence recognition. In S.Gielen and B.Kappen (eds.), *Proc. ICANN'93. International Conference on Artificial Neural Networks*, pages 404–407, Springer, 1993.

[28] D.L. James and R. Miikkulainen. SARDNET: a self-organizing feature map for sequences. In G.Tesauro, D.Touretzky, and T.Leen (eds.), *Advances in Neural Information Processing Systems 7*, pages 577-584, MIT Press, 1995.

[29] J. Kangas. Time-delayed self-organizing maps. In: *Proc. IEEE/INNS Int. Joint Conj. on Neural Networks 1990*, 2, pages 331-336, 1990.

[30] S. Kaski. Bankruptcy analysis with self-organizing maps in learning metrics. *IEEE Transactions on Neural Networks*, 12:936-947, 2001.

[31] T.Kohonen. *Self-organizing maps*. Springer, 1997.

[32] T. Kohonen. Learning vector quantization. In M. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 537–540. MIT Press, 1995.

[33] T. Koskela, M. Varsta, J. Heikkonen, and K. Kaski. Recurrent SOM with local linear models in time series prediction. In M.Verleysen (ed.), *6th European Symposium on Artificial Neural Networks*, pages 167–172, De facto, 1998.

[34] T. Kohonen, S. Kaski, K. Lagus, and T. Honkela. Very large two-level SOM for the browsing of newsgroups. In C. von der Malsburg, W. von Seelen, J. C. Vorbrüggen, and B. Sendhoff (eds.), *Proceedings of ICANN96*, pages 269–274, Springer, 1996.

[35] C.-M. Kuan and K. Hornik. Convergence of learning algorithms with constant learning rates. *IEEE Transactions on Neural Networks* 2(5):484-489, 1991.

[36] H. J. Kushner and D. S. Clark. *Stochastic approximation methods for constrained and unconstrained system*s. Springer, 1978.

[37] T. Martinetz, S. Berkovich, and K. Schulten. "Neural-gas" network for vector quantization and its application to time-series prediction. *IEEE-Transactions on Neural Networks* 4(4): 558-569, 1993.

[38] T. Martinetz and K. Schulten. Topology representing networks. *Neural Networks* 7(3):507-522, 1993.

[39] S. C. Kremer. Spatio-temporal connectionist networks: A taxonomy and review. *Neural Comp.*, 13(2):249–306, 2001.

[40] B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Tansactions on Neural Networks*, 6(5):1212–1228, 1995.

[41] H. Ritter. Self-organizing maps in non-euclidean spaces. In E. Oja and S. Kaski, editors, *Kohonen Maps*, pages 97–108. Springer, 1999.

[42] H. Ritter, T. Martinetz, and K. Schulten. *Neural computation and self-organizing maps, an introduction.* Addison-Wesley, 1992.

[43] A. Sadeghi. *Asymptotic behaviour of self-organizing maps with non-uniform stimuli distribution.* Technical Report, Universität Kaiserslautern, FB Mathematik, Germany, July 1996.

[44] A. S. Sato and K. Yamada. Generalized learning vector quantization. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 423–429. MIT Press, 1995.

[45] J. Sinkkonen and S. Kaski. Clustering based on conditional distribution in an auxiliary space. *Neural Computation*, 14:217-239, 2002.

[46] A. Sperduti. Neural networks for adaptive processing of structured data. In: G.Dorffner, H.Bischof, K.Hornik (eds.), *ICANN'2001*, pages 5-12, Springer, 2001.

[47] A. Sperduti and A. Starita. Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks*, 8(3):714-735, 1997.

[48] J. Vesanto. Using the SOM and local models in time-series prediction. In: *Proc. Workshop on Self-Organizing Maps 1997*, pages 209-214, 1997.

[49] T. Villmann, R. Der, M. Herrmann, and T. Martinetz. Topology preservation in self-organizing maps: exact definition and measurement. *IEEE Transactions on Neural Networks* 8(2):256–266, 1997.

[50] T. Voegtlin. Context quantization and contextual self-organizing maps. In: *Proc. Int. Joint Conf. on Neural Networks*, vol.5, pages 20-25, 2000.

[51] T. Voegtlin. Recursive self-organizing maps. *Neural Networks* 15(8-9):979-992, 2002.

[52] T. Voegtlin and P.F. Dominey. Recursive self-organizing maps. In N.Allison, H.Yin, L.Allinson, and J.Slack (eds.), *Advances in Self-Organizing Maps*, pages 210-215, Springer, 2001.

Appendix

Derivative of f for VQ

$\partial f(\tilde{d}(a_k, N))/\partial \tilde{d}(a_k, n_j)$ for VQ is to be computed. Note that

$$\chi(T_i, n_j) = \mathrm{H}\left(\sum_l \mathrm{H}(\tilde{d}(T_i, n_l) - \tilde{d}(T_i, n_j)) - N + 1.5\right)$$

where H is the Heaviside function

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The derivative of H is the function δ which is a symmetric function which is non-vanishing only for $x = 0$. Hence we find

$$\frac{\partial f(\tilde{d}(T_i, N))}{\partial \tilde{d}(T_i, n_l)} = \frac{1}{2}\sum_j \frac{\partial \chi(T_i, n_j)}{\partial \tilde{d}(T_i, n_l)} \cdot \tilde{d}(T_i, n_j) + \frac{1}{2}\sum_j \chi(T_i, n_j) \cdot \frac{\partial \tilde{d}(T_i, n_j)}{\partial \tilde{d}(T_i, n_l)}$$

The second term yields the standard Hebb factor of VQ, i.e. it yields the contribution $\frac{1}{2}\chi(T_i, n_l)$ to the above general formulas for the derivatives. The first summand equals

$$\frac{1}{2}\sum_{j,k} \delta(\tilde{d}(T_i, n_k) - \tilde{d}(T_i, n_j)) \left(\frac{\partial \tilde{d}(T_i, n_k)}{\partial \tilde{d}(T_i, n_l)} - \frac{\partial \tilde{d}(T_i, n_j)}{\partial \tilde{d}(T_i, n_l)}\right) \cdot \\ \delta\left(\textstyle\sum_k \mathrm{H}(\tilde{d}(T_i, n_k) - \tilde{d}(T_i, n_j)) - N + 1.5\right) \tilde{d}(T_i, n_j)$$

Hence it vanishes because δ is symmetric, and the terms involving δ are nonvanishing iff the role of $\tilde{d}(T_i, n_k)$ and $\tilde{d}(T_i, n_j)$ can be changed.

Derivative of f for approximate SOM

The derivative we are interested in can be computed by

$$\frac{1}{2}\sum_j \frac{\partial \chi_S(T_i,n_j)}{\partial \tilde{d}(T_i,n_l)} \sum_k \eta(nh(n_j,n_k))\tilde{d}(T_i,n_k) \\ +\frac{1}{2}\sum_j \chi_S(T_i,n_j)\sum_k \eta(nh(n_j,n_k))\frac{\partial \tilde{d}(T_i,n_k)}{\partial \tilde{d}(T_i,n_l)}$$

The second term yields the usual SOM-update with slightly different notion of the winner as explained in Section 5. The contribution to the above formulas is $\frac{1}{2}\sum_j \chi_S(T_i,n_j)\eta(nh(n_j,n_l))$. The first term vanishes, as can be seen as follows: we use the identity

$$\chi_S(T_i,n_j) = \mathrm{H}\left(\sum_l \mathrm{H}\left(\sum_k \eta(nh(n_l,n_k))\tilde{d}(T_i,n_k) \right.\right. \\ \left.\left. - \sum_k \eta(nh(n_j,n_k))\tilde{d}(T_i,n_k)\right) - N + 1.5\right)$$

Hence the first sum equals

$$\frac{1}{2}\sum_{j,o} \delta\left(\sum_s \mathrm{H}\left(\sum_t \eta(nh(n_s,n_t))\tilde{d}(T_i,n_t) - \sum_t \eta(nh(n_j,n_t))\tilde{d}(T_i,n_t)\right)\right. \\ \left. -N+1.5\right)\cdot \\ \delta\left(\sum_t \eta(nh(n_o,n_t))\tilde{d}(T_i,n_t) - \sum_t \eta(nh(n_j,n_t))\tilde{d}(T_i,n_t)\right)\cdot \\ \sum_k \tilde{d}(T_i,n_k)\eta(nh(n_j,n_k))\left(\eta(nh(n_o,n_l)) - \eta(nh(n_j,n_l))\right)$$

This vanishes because δ is symmetric and the above δ-terms are nonvanishing only if the weighted distances of n_o and n_j can be substituted.

Derivative of f for NG

The derivative is

$$\frac{1}{2}\sum_j \frac{\partial \eta(rk(i,j))}{\partial \tilde{d}(T_i,n_l)}\tilde{d}(T_i,n_j) + \frac{1}{2}\sum_j \eta(rk(i,j))\frac{\partial \tilde{d}(T_i,n_j)}{\partial \tilde{d}(T_i,n_l)}$$

where the second term yields the usual update rule for NG, i.e. the contribution $\frac{1}{2}\eta(rk(i,l))$ to the above formulas. The first term vanishes, which can again be explicitely computed by using the identity

$$rk(i,j) = \sum_k H(\tilde{d}(T_i,n_j) - \tilde{d}(T_i,n_k))$$

Then the first term becomes

$$\frac{1}{2}\sum_{j,k}\frac{\partial\eta(rk(i,j))}{\partial rk(i,j)}\delta(\tilde{d}(T_i,n_j) \\ - \tilde{d}(T_i,n_k))\tilde{d}(T_i,n_j)\left(\frac{\partial\tilde{d}(T_i,n_j)}{\partial\tilde{d}(T_i,n_l)} - \frac{\partial\tilde{d}(T_i,n_k)}{\partial\tilde{d}(T_i,n_l)}\right)$$

This vanishes due to the properties of δ.